University of Ljubljana
Faculty of Computer and Information Science

Marko Čeferin

Specification and Implementation of a Light-Weight Internet Content Delivery Platform

diploma thesis

First cycle professional study programme
computer and information science

Mentor: Assist. Prof. PhD Veljko Pejović

Ljubljana, 2017

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Marko Čeferin

Specifikacija in implementacija preproste internetne platforme za prenos vsebin

diplomsko delo

Visokošolski strokovni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Veljko Pejović

Ljubljana, 2017

Faculty of Computer and Information Science issues the following thesis: Specification and Implementation of a Light-Weight Internet Content Delivery Platform

Thesis topic:

World Wide Web (the Web) is possibly the largest and the most important information space on Earth. The Web's growth was facilitated by initially simple hypertext language (HTML) and the transfer protocol (HTTP). However, decades of development have led to a number of features and proprietary vendor extensions bundled to the original specifications, significantly increasing the complexity of the Web. Recently, computing has evolved from mainframes and desktop PCs to mobile personal devices, wearable computers and a whole ecosystem of small pervasive computation and communication devices connected into the Internet of Things (IoT). However, the complexity of today's HTTP and HTML prevents many of the low-resource computers present in the current heterogeneous landscape of devices from running modern Web browsing engines and benefiting from, as well as contributing to, the information on the Web. In this thesis topic we call for design and implementation of a novel light-weight markup language and a transfer protocol for hyperlinked information. The key property of the solution should be to, unlike the current Web, separate the content from the presentation, and consequently afford seamless information parsing, increased human readability of the content, and reduce the implementation complexity of content browsing engines.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:
Specifikacija in implementacija preproste internetne platforme za prenos vsebin

Tematika naloge:

Svetovni splet (splet) je verjetno največji in najbolj pomemben informacijski prostor na Zemlji. Rast spleta je bila omogočena s sprva enostavnim hipertekstovnim jezikom (HTML) in protokolom za prenos podatkov (HTTP). Toda desetletja razvoja so privedla do več nestandardnih zasebnih razširitev originalnih specifikacij, kar je bistveno povečalo kompleksnost spleta. Nedavno se je računalništvo razvilo iz osrednjih *(mainframe)* ter namiznih računalnikov v prenosne naprave, nosljive računalnike in celoten ekosistem majhnih vseprisotnih računskih ter komunikacijskih naprav povezanih v Internet stvari *(Internet of Things)*. Zaradi kompleksnosti današnjih HTTP in HTML veliko manj zmogljivih računalnikov, prisotnih v trenutnem raznolikem okolju naprav, ni zmožno uporabljati modernih spletnih brskalnikov ter uporabljati ali prispevati informacije na spletu. V temi te diplomske naloge pričakujemo zasnovo in implementacijo novega preprostega formata za označevanje besedila in protokola za prenos hipertekstovnih informacij. Ključen cilj rešitve naj bi bilo to, da za razliko od trenutnega spleta loči vsebino od prezentacije in s tem omogoči enostavno razčlenjevanje, povečano človeško berljivost vsebine in zmanjšano kompleksnost implementacije brskalnika vsebine.

I would like to express my gratitude to my mentor Veljko Pejović for the useful comments, remarks and advice while working on this thesis.

I would also like to express my thanks to the company Beyond Semiconductor for support while I was working on the thesis.

Contents

Abstract

Title: Specification and Implementation of a Light-Weight Internet Content Delivery Platform

Author: Marko Čeferin

Abstract: The World Wide Web is the largest and most popular hypertext information system in the world. It is used by billions of people primarily to share information between themselves. As more and more features are added to it, modern Web has become a powerful general-purpose application platform. The cost of that complexity is the content itself being obscured behind all the layout, presentation and interaction. This thesis specifies a simple protocol and document format intended to solve the problem of exceeding complexity of using the Web to access content sources without using complicated monolithic applications that require powerful hardware to run. This provides an alternative to the current Web technologies for browsing pages focused on the content itself, while being significantly simpler to implement and use. By using the new technologies, accessing content sources becomes simple, while the current Web remains useful as an application platform. Foremost, this simplifies creation of new browser engines that are easier to implement and need less processing resources to run. At the same time, it enables programmatic access to content that used to be accessible only to human readers.

Keywords: Web, protocol, document format, content delivery, semantic markup.

Povzetek

Naslov: Specifikacija in implementacija preproste internetne platforme za prenos vsebin

Avtor: Marko Čeferin

Povzetek: Svetovni splet *(World Wide Web)* je največji in najbolj popularen hiperterkstovni informacijski sistem na svetu. Milijarde ljudi ga uporablja za dostop do informacij. S stalnim dodajanjem zmogljivosti je splet postal vsenamenska aplikacijska platforma. Posledica te kompleksnosti je to, da je sama vsebina pogosto skrita za predstavitvijo in interakcijo. To diplomsko delo definira specifikacije za protokol ter format dokumentov, ki poskuša rešiti problem kompleksnosti pri uporabi spleta za dostop do vsebinskih virov. To predstavlja alternativo uporabi trenutnih spletnih tehnologij za brskanje po straneh s poudarkom na vsebini, ki je bistveno bolj enostavna za implementacijo ter uporabo. S tem postane dostop do vsebinskih virov enostaven, splet pa se še vedno lahko uporablja kot aplikacijska platforma. Na ta način se predvsem poenostavi izdelava novih spletnih brskalnikov, ki so lažji za implementacijo ter potrebujejo dosti manj računalniških virov. Hkrati se omogoči enostaven programski dostop do vsebin, ki so bile klasično dostopne le ljudem.

Ključne besede: Splet, protokol, format dokumentov, dostop do vsebin, semantično označevanje besedila.

Razširjen povzetek

Svetovni splet *(World Wide Web)* je najbolj popularen informacijski prostor na svetu. Milijarde ljudi ga uporablja za dostop do informacijskih virov, ki so ponavadi sestavljeni iz besedila, slik in videa.

Glavna sestavna dela spleta sta protokol za prenos vsebin HTTP ter format dokumentov HTML, pri čemer slednji tudi uporablja stilske podlage CSS za oblikovanje spletnih strani ter programski jezik JavaScript za implementacijo interaktivnih vsebin.

Uporabniški vmesnik do spleta, sestavljenega iz teh gradnikov, je spletni brskalnik. To je program, ki lahko dostopa do spletnih strani preko protokola HTTP, postavi vsebino HTML in izriše stran, oblikovano s CSS, za tem pa zažene priložene JavaScript programe. S tem se uporabniku izriše končna oblika spletne strani, s katere lahko razbere informacije.

Toda skozi desetletja razvoja je splet pridobil veliko različnih funkcij. Standardi so se razvijali, popularni spletni brskalniki pa dodajali svoje nestandardne razširitve, ki so se kasneje pogosto uveljavile kot vseprisotne in so občasno postale tudi del standardnih specifikacij. Splet, ki je sprva bil le medij za prenos statičnih hipertekstovnih informacij, je postal vsenamenska aplikacijska platforma, na kateri se lahko postavijo vse vrste aplikacij, ki so prej bile omejene le na lokalne programe.

S temi zmožnostmi je splet postal izjemno kompleksen. Kar je nekoč bilo mogoče implementirati v razumnem času je danes tako komplicirano, da kljub popularnosti spleta obstajajo le štiri različne popolne implementacije jeder spletnih brskalnikov: Gecko (Mozilla Firefox), Blink (Google Chrome,

Opera), WebKit (Apple Safari) in EdgeHTML (Microsoft Edge). Vsaka od teh implementacij je velik monoliten program, ki potrebuje dokaj moderen in zmogljiv procesor ter veliko količino sistemskega pomnilnika za delovanje. Obstajajo tudi bolj preprosti spletni brskalniki, ampak niso sposobni prikazati vsebine na vsaki spletni strani. Spletnih strani, ki potrebujejo podporo najnovejših ter najzahtevnejših tehnologij, je čedalje več. Zaradi tega je dostop do spleta skoraj nemogoč velikemu številu naprav, od manj zmogljivih vgrajenih naprav, ki so del Interneta stvari, do par let starih pametnih telefonov, ki niso več sposobni zaganjati spletnih brskalnikov, saj slednji sedaj potrebujejo več sistemskih virov.

Drugi del problema spleta je to, da je sama vsebina spletnih strani (večinoma tekst in slike) skrita za postavitvijo strani (ki je prepletena z vsebino v HTML), oblikovanjem strani preko CSS (kar je pogosto nujno za pravilno razumevanje vsebine) in izvedbo interaktivnih aplikacij (ki omogočijo prikaz vsebine). Zaradi tega veliko spletnih strani ni dostopno na brskalnikih, ki ne podpirajo teh tehnologij, čeprav so drugače sposobni prikazati samo vsebino. Posledično to tudi pomeni, da je vsebina spletnih strani težko dostopna programsko. Spletne strani, ki so narejene tako, da so berljive le človeškim uporabnikom, ki do njih dostopajo na kateri izmed podprtih naprav (pogosto le navadni osebni računalniki in pametni telefoni), so pogosto nedostopne spletnim iskalnikom, orodjem za prikaz vsebine na bolj omejenih napravah (na primer bralnikih elektronskih knjig) in podobno. HTML5 vsebuje podporo za zapis vsebin izključno s semantičnimi elementi, toda to pomeni le to, da je programski dostop možen le do spletnih strani, ki se držijo strogih pravil uporabe teh elementov, medtem, ko je veliko drugih strani še vedno povsem nedostopnih in zahteva implementacijo kompleksnih spletnih brskalnikov.

Zaradi pomanjkljivosti prvotnih specifikacij tehnologij, nestandardnih razširitev spleta, ki so postale vseprisotne, ter uporabniških napak, ki so jih spletni brskalniki sami poskušali popraviti, je za popolno implementacijo tehnologij spleta potrebno poskrbeti, da so podprti različni načini za obdelavo napačno strukturiranih dokumentov ter nestandardne uporabe različnih

funkcionalnosti. To doda bistveno kompleksnost celoti spleta in oteži dostop do takih dokumentov z brskalniki, ki ne podpirajo enakih nestandardnih funkcij.

To diplomsko delo predstavi specifikacijo novega protokola za prenos vsebin in formata dokumentov, katerih glavni cilji so preprostost implementacije, nedvoumnost sintakse, poudarek na semantični strukturi, brez mešanja oblikovanja z vsebino, in človeški berljivosti formata dokumentov. Ker ni le razširitev že obstoječih tehnologij, ni potrebno, da so te tehnologije podprte, s čemer se zmanjša kompleksnost implementacije ter omogoči bolj smiselna zasnova funkcionalnosti.

Nov protokol za prenos vsebin je podoben protokolu HTTP, pri čemer poenostavi in posploši sintakso in različne funkcije protokola. Čeprav ne podpira vseh zmogljivosti, ki jih ima HTTP, ima podporo za pomembne funkcije za dostop do vsebinskih virov in je dosti manj kompliciran za implementacijo in uporabo. Odpravi tudi nekaj pomanjkljivosti protokola HTTP, kot so uporaba neidealnega kodiranja besedila, ki ga HTTP zahteva, in zapis imen datotek, ki vsebujejo posebne znake. Med ostalim pa tudi podpira novo funkcijo, ki omogoča dostop do delov vsebinskih virov s pomočjo semantičnih zahtev, tako, da se lahko prenese le del dokumenta glede na sekcije. Sintaksa protokola je strogo definirana, kar pomeni, da je manj verjetno, da bodo implementacije napačne, medtem, ko preprostost protokola pomeni, da bo več implementacij popolnih ter bo manj dela z odkrivanjem, katere funkcionalnosti odjemalec in strežnik podpirata.

Nov format dokumentov zavrže mešanje postavitve delov spletne strani z vsebino, ki se uporablja v HTML. Format zapisa dokumenta je zasnovan na vrsticah besedila z uporabo zamikov in je posledično bolj berljiv tudi brez uporabe spletnega brskalnika. Zapis oblikovanega besedila je zasnovan na enostavnih jezikih za označevanje besedila, kot je npr. Markdown, kar tudi pripomore k lažjem razumevanju sintakse. Označevanje besedila je tudi manj občutljivo na človeške napake, saj za razliko od HTML ne zahteva, da se posamezni formati končajo v nasprotnem vrstnem redu od tega, kako so bili

uporabljeni.

Vsak element strukture dokumenta ima semantični pomen. Format dokumentov ne podpira zapisa oblike in interakcije, saj je to prepuščeno uporabniškemu vmesniku. To pomeni, da se lahko vsebina dokumenta prilagodi uporabnikovi napravi in se zato lahko brez posebne postavitve strani prikaže pravilno na vseh napravah, od osebnih računalnikov do bralnikov spletnih knjig ali pa celo na majhnih zaslonih z omejeno velikostjo. Naenkrat je vsebina tudi programsko dostopna, saj se ni potrebno ukvarjati s tem, kateri element je zares vsebina, kateri pa je le del strukture in oblike spletne strani. Sama oblika ter ostali metapodatki so implicitno definirani preko semantičnih elementov. Na primer, poudarjeno besedilo se lahko izriše z odebeljeno pisavo, kazalo se lahko avtomatsko sestavi glede na imena sekcij, in tako naprej.

Zaradi tega so lahko implementacije nove tehnologije bistveno bolj preproste od trenutnih implementacij spleta in naenkrat zmožne dostopa do vseh strani, zgrajenih s pomočjo teh tehnologij. Čeprav format dokumentov ne podpira oblikovanja, to še ne pomeni, da mora biti stran brez stilov, saj lahko uporabniški vmesnik uporabi svoje oblikovanje po željah uporabnika. Ker je struktura dokumentov semantična in vsebina ni odvisna od oblikovanja in interaktivnih aplikacij, so dokumenti dostopni tudi orodjem brez potrebe po človeškem bralcu, ki vizualno razbere vsebino iz izrisane strani. Zaradi preprostih implementacij, ki ne zahtevajo kompliciranih zmožnosti in računalniških virov, je ta informacijski prostor bolj dostopen tudi manj zmogljivim napravam in Internetu stvari.

Tudi če se celotna tehnologija ne uveljavi in končni uporabniki ne bodo imeli programske opreme za uporabo tehnologije, se lahko vsaj format dokumentov uporabi za gradnjo strani na spletu, pri čemer se prevede v HTML na podoben način, kot to počnejo obstoječi enostavni jeziki za oblikovanje besedila, kot je npr. Markdown. S pravilno uporabo je izvorni dokument še vedno preprost in končni rezultat semantični HTML dokument, čeprav zahteva navaden spletni brskalnik za uporabo.

Chapter 1

Introduction

The World Wide Web is the most popular hypertext information system in the world. It is used by billions of people primarily to share information between themselves, most often in the form of retrieving pages composed from text, images and videos from a publishing website. It is powered by a text-based request-response protocol HTTP [6, 13, 14] and a document markup format HTML [5] transferred over it.

This thesis specifies a content transfer protocol and a document format aimed at replacing that part of the Web.

1.1 Motivation

Due to decades of feature expansion, the current Web is a content delivery and application platform of vast complexity. It supports various features, including the core HTML-based document contents and layout, styling through deprecated HTML attributes, advanced CSS [7] styling, scripting using ECMAScript [11] (JavaScript), vector graphics using SVG [12], mathematical notation via MathML [20] and much more.

However, that has resulted in the entire Web suite being so complicated that only a handful of up-to-date implementations of Web browsing engines exist (currently, Gecko powering Mozilla Firefox, Blink powering Google

Chrome and Opera, WebKit powering Apple Safari and EdgeHTML powering Microsoft Edge). Implementing a new Web browser from scratch takes tremendous effort of many people and takes an enormous amount of development time and is seldom done these days. All modern Web browsers are also very resource-intensive [24, 25, 22], often requiring a modern processor and hundreds of megabytes of memory in order to function properly. The Web is getting increasingly more complex and tends to push against the limits of hardware, with older and less powerful devices unable to keep up with it. Considering the current trend towards smaller pervasive computing devices, the Web is becoming less practical and accessible. Smaller devices, such as embedded computers or IoT devices, have never been compatible with any modern browsers. While browsers with lesser system requirements exist, they usually do not support all modern Web technologies and cannot display content from websites that rely on them.

Many pages rely on the user-agent being a modern resource-heavy Web browser running on one of the supported device types (usually either a personal computer with a large screen or a modern handheld device, such as a smartphone) being accessed by a human user using the standard human-machine interface methods (usually keyboard and mouse or a smartphone touchscreen).

Besides that, Web pages often obfuscate the actual content (text and images, in most cases) behind page layout, styling and interactive scripts. **Extracting the content often requires a human reading the page and the content can be very hard, if not near-impossible, to extract without human interaction**. Far too often, Web pages rely on style-based element modifications that may change or even hide elements that might otherwise seemingly appear to be content. When pages are dynamically generated via scripts, a proper Web browser is required to render the final page in order to access its content, and even then, extracting the content without the surrounding layout, navigation and other miscellaneous elements may be nontrivial. By 2020, 20 [16] to 30 [1] billion devices are estimated to

be connected to the Internet, so most of the communication will not involve a human person. Together with further intelligence pushed into these devices, there is an obvious need for them to be able to "read" the Web.

While all of that enables the Web to support various kinds of interactive applications and other complex pages, a significant part of Web usage remains accessing static pages containing content composed of text and embedded media (images, videos, and others) or at least pages that could be rewritten as such without missing out on the content. However, accessing the content on websites still often includes loading, rendering and executing all the layout, styles and scripts, taking up a significant amount of resources. Not only does displaying text on a modern Web page require a relatively powerful processor and a significant amount of memory, it can also require hundreds of HTTP requests for various linked styles, scripts and dynamically accessed data, which can take numerous network round-trips and a notable amount of bandwidth, **often being undesirable on limited-bandwidth or high-latency connections, such as mobile or satellite Internet connections**.

Based on the above observations, we conclude that the Web, while certainly powerful, has become far too complex for accessing simple documents. A significant portion of Web usage consists of the users merely wanting to see the static text, picture and video content, not the styling or interaction provided alongside (and especially not scripts whose sole purpose is gathering information about the user or displaying advertisements at the cost of the user's privacy, bandwidth and system resources). The Web could benefit from separating the core content delivery systems from the entire application framework.

This thesis tries to specify a new request-response networking protocol for content delivery and a light-weight document markup format with a focus on simplicity of implementation, precise specification of syntax and features, human readability and focus on the semantic part of document contents at the expense of layout, styles and interactivity. The goal of the protocol

is to provide functionality equivalent to a subset of HTTP that deals with retrieving content from servers. The protocol should be well-defined to avoid incompatibility and easy to implement so that implementations support all of it, rather than a subset. The markup format should focus on the content with semantic markup. It should not define layout or styles, since that can be handled by the user-agent in a device-specific manner. For example, the user-agent could adjust the text to fit on the device's screen or change colors based on the environment in which the content is accessed, e.g. indoor vs outdoor, as sensed by the device's ambient light sensor, without having to rely on the document's style and scripts doing that. The markup should be simple to both write by humans and parse by computers in order to avoid mistakes or incorrect implementations. To aid programmatic use and improve performance on low-bandwidth connections, it should be possible to perform partial content transfer and retrieve only specific sections of document contents using semantic selectors on the protocol level.

The primary reason for new specifications is that while it would be possible to use a subset of HTTP and semantic HTML5 that are simple and well-defined, websites relying on the complex features would still exist. By starting from scratch, we can ensure that all websites using these new technologies will be limited to their scope instead of being nearly indistinguishable from complicated Web sites. That way, every page will be accessible using any user-agent built to support these new technologies instead of attempting to open Web sites that supposedly use a limited subset of current Web technologies and having to handle the cases where other features are also used. Additionally, by starting from scratch, decades-old mistakes that still persist in the Web due to their ubiquity can be avoided at the design phase.

1.2 Contributions

The primary contribution of this thesis are the specifications for the new protocol, *CNP*, and document markup format, *CNM*. Together, they define

a new content delivery platform and information space named *ContNet*. The protocol allows content delivery using an easy-to-implement platform without having to worry about supporting every intricacy of existing protocols, such as HTTP or FTP. The document format provides a way to write semantic documents in a simple and readable syntax that's also easy to parse programmatically. The specifications can be used for improved versions of the protocol and document format or serve as a basis for similar implementations.

A secondary contribution is a free and open source implementation of the specifications. It primarily includes:

- a server/client library for the new protocol [32]
- server serving files over the new protocol [33]
- a parser/composer library for the new document format [31]
- a gateway that allows accessing the new protocol and document format via HTTP and HTML [34]

Even if the new content-delivery platform is not used in its entirety, the implementations can still be used to serve standard HTTP+HTML Web pages from documents written in the new document format.

Chapter 2

Background and related work

The Web, specifically HTTP as the protocol and HTML (with the use of CSS for styling and JavaScript for scripting) as the document format, is the de-facto hypertext information system implementation and the largest content distribution platform in the world. Since the Web's inception, it has constantly been in development, with many new features being added both in official specifications and as proprietary vendor extensions. The most popular Web browsers often added features on their own, which were sometimes adapted by other browsers and turned into an unofficial standard. Nowadays, the functionality of the Web is primarily defined by the main implementations, with new standards often reflecting that in turn. The current Web houses a powerful portable application platform that can safely run various programs that are not limited only to displaying static content, such as document editors, real-time chat platforms, video games and even applications that facilitate content delivery in other formats than those natively supported by the Web.

2.1 HTTP

HTTP is a protocol used by user-agents such as Web browsers to request content from Web servers. In its core, HTTP is a request-response protocol. It

defines request and response messages containing a method line that describes the meaning of the message and a list of headers following it. It also defines a significant amount of different status codes to be used in the response method line, as well as numerous request and response headers serving different purposes.

The second version of the protocol, HTTP/2 [4], is significantly different, being a multiplexed binary protocol based on data frames instead of a text-based request-response protocol. Being more complicated and containing more features and optimizations than HTTP/1.X, it is designed to solve various problems prominently caused by the new requirements of the Web as an application platform, such as using one multiplexed connection to avoid both head-of-line blocking and doing a large number of round-trips when retrieving page resources (mostly images, stylesheets and scripts).

By being binary instead of text-based, it avoids many problems caused by improper text-based parsing of protocol messages at the cost of the new protocol messages not being readable or writable by humans. However, the basic operation is still based on HTTP/1.X, which means that it inherits several flaws from it. For example, on HTTP/2, cookies share all the same flaws [30] as on HTTP/1.X.

2.2 HTML

HTML describes documents composed of text, images, layout and various features. It integrates content and layout into a single markup. When mixed with CSS and JavaScript, especially when the content is loaded dynamically using scripts, the page has to fully render to draw the content on the screen, with the content only being presented in a format meant to be consumed by humans.

HTML version 4 [18] separated the styling of the document into separate CSS stylesheets, but the document still contained intertwined layout and content. HTML5 [23] defined a collection of semantic elements that, when

used properly, tend to result in a document whose content is programmatically accessible, but that requires the page designer to actually use these features properly. Modern HTML is used as the basis for building arbitrary Web applications by using complex CSS styling and live manipulation of the *Document Object Model* using JavaScript, while at the same time containing a semantic subset suitable for use as content markup. More often than not, a significant portion of the feature set is used in websites, resulting in the need to run the entire Web application to access the actual content.

2.3 Gopher

The Gopher [2] protocol is a predecessor to the World Wide Web. Instead of relying on a standard document format, it primarily consists of menus containing links to other pages, which may be other menus or a file with content, but not both. The protocol defines specific file types using a single character identifiers; this system predates MIME file types commonly seen in email and the Web and is more primitive and harder to extend. Due to that, the primary document type of Gopher pages is plain text, though non-canonical extensions, such as an entry for the HTML file type, exist.

Gopher tries to imitate the file system layout more closely than the Web, with menus being roughly equivalent to directory listings. While the Web can contain directory listings, it has no specific semantic constructs that would signify their meaning globally, without relying on each individual website's custom structure.

While Gopher is significantly simpler than the Web, it is also more limited and unable to describe common structured and formatted text used as the basis for many documents, since it is mostly limited to menus and text. If extensions, such as support for HTML pages, are used, the complexity extends by the significant amount that HTML introduces.

Chapter 3

Simple content-delivery protocol

The *ContNet Protocol*, *CNP*, is a request-response application protocol meant to facilitate hypertext content requests and delivery over the Internet. It is designed to be an alternative to a part of HTTP; specifically, the subset dealing with static content (text, images, videos and similar). The protocol is designed to be simple, easy to implement and unambiguous, as well as strict in order to avoid having to support multiple incorrect implementations. CNP uses the client-server computing model. A user-agent sends a CNP request to a server, which then replies with the CNP response containing the status of the request and possibly the requested resource as the message body.

The complete protocol specification is available in Appendix A.

3.1 Syntax

Technically, CNP is a binary protocol. The message header is a line of ASCII text, but is defined as byte sequences that the text represents. Due to that, it does not permit any letter case or whitespace insensitivity, which makes it strict and less likely to be implemented and parsed incorrectly, as long as implementations do not treat it as a text protocol. This should also

avoid problems with text encodings; the protocol header does not use any encoding, since it is defined using ASCII bytestrings. Thus, using incompatible serialization of the text that is a part of the header will result in an error.

Both requests and responses are instances of CNP messages and share the same syntax. A CNP message consists of a header and optional body contents. The header contains the protocol version as the prefix, which can be used to select the proper parsing method, if incompatible syntax is introduced in the future. The version is followed by the message intent, which is an arbitrary string meant to contain the meaning of the message; the intent is roughly equivalent to HTTP request paths and response status codes. After the intent comes an arbitrary number of parameters, represented as key-value pairs. The parameters are used to facilitate additional functionality, such as caching, metadata or content selectors.

The header fields are separated using a single space as a delimiter, with the parameter key and value being joined by an equals sign. The entire header ends with a line feed byte; unlike HTTP, CNP does not use a carriage return and line feed sequence to delimit the header, which makes parsing simpler, since only a single byte needs to be matched. An example message header is provided in listing 3.1.

```
1 cnp/0.4 message_intent parameter1=value1 parameter2=value2
```

Listing 3.1: CNP message header

All strings in the header (intent and parameter keys/values) are escaped by replacing specific bytes with a 2-byte sequence. The zero byte is replaced by `\0`, line feed by `\n`, space by `\_`, equals sign by `\-` and backslash by `\\`. That way, there are no spaces or equals signs in the document other than the ones required by the syntax. Thus, the message can be split into fields by splitting on the space byte, then parameters can be split on the equals sign byte.

CNP message syntax is also defined by an EBNF grammar, shown in listing 3.2.

```
raw_character = ? any byte except "\0", "\n", " ", "=", "\\" ? ;
escaped_character = "\\" ( "0" | "n" | "_" | "-" | "\\" ) ;
ident_character = raw_character | escaped_character ;
identifier = ident_character, { ident_character } ;
number = "0" | ( "1" ... "9" ), { "0" ... "9" } ;

version = "cnp/", number, ".", number ;
intent = identifier ;
parameter = [ identifier ], "=", [ identifier ] ;

header = version, " ", intent, { " ", parameter }, "\n" ;
body = { ? any byte ? } ;

message = header, body ;
```

Listing 3.2: CNP message EBNF grammar

3.2 Request and response

To retrieve content over CNP, a client connects to a server and sends a request to it. Based on the resource identified by the requested path provided in the request intent header field and any parameters that may affect the retrieved content, the server sends a response message with the content, if any, attached as the message body. The client then reads that response and handles it based on its intent field. An example request using several parameters with a corresponding response and the retrieved content is shown in figure 3.1.

```
cnp/0.4 contnet.org/spec/cnp0.4/ if_modified=2017-09-07T17:07:36Z select=byte:-64


cnp/0.4 ok length=65 modified=2017-09-11T03:17:11Z select=byte:0-64 type=text/cnm
title
    ContNet Protocol specification, version 0.4 (2017-09-04)
```

Figure 3.1: Example CNP request (above) and response with content (below)

The request and response messages are both instances of CNP messages.

They differ primarily in the meaning of the message intent field and the header parameters.

The request uses the server's hostname and a filepath as the message intent. That corresponds to the host and path parts of a URL and defines which resource on the server the client wants to access. Since the request intent is not actually a URL, it does not requite implementing an entire URL parser; instead, after the CNP message is decoded, the only operation necessary to retrieve the host and path strings is splitting the request intent before the first slash byte in it.

The request parameters mostly define metadata of the request body content, if any, such as length, filename and content type. Notable request parameters are `if_modified`, which facilitates caching content, and `select`, which is used for selectors described in Section 3.4.

3.3 Parsing

CNP messages are very easy to parse. Since the content of the message header values cannot contain any syntactically relevant characters unencoded, it is generally safe to use simple bytestring splitting algorithms to decode message headers.

The simplest method to parse a CNP message is to use batch parsing: first, read the entire message header (until the first line feed) from the connection, then use a sequence of bytestring manipulation functions on the entire header. First, split the header line on every space byte. The result of that should be an array of two or more fields; if less, then the message was not valid. The first field is the CNP version, second is the message intent and the remainder are parameters. Each parameter can be split on the equals byte; if that does not result in exactly two fields (key and value), then the parameter was invalid. After all of that is done, the intent and all parameter keys and values have to be unescaped, which can be done using a sequence of bytestring substitutions. Any remaining data on the connection is the message body.

Alternatively, stream parsing can be used to parse a message while streaming it without having to read the entire header first. First, read a field that ends with a space (version); if a newline is found instead, the message is invalid, since it lacks an intent. Then, read one or more fields that either end in space or newline, with the latter ending the entire header. The fields will be the message intent and parameters. As each of these fields is read, it can also be decoded on-the-go by looking for a backslash byte and replacing it and the following character with the proper unescaped byte. For parameters, the key and value can be read separately by first reading until an equals sign (with a stray space or line feed marking the header as invalid), then until a space or line feed.

All of these parsing operations are exact and simple. The field separator is always exactly one space byte, never multiple spaces or e.g. tab bytes instead of spaces. The end of header is always a line feed; the parser does not have to consider special cases where a carriage return is present alongside or instead of it. Parsing does not require any lookahead and can be implemented as a simple finite state machine operating on individual bytes. A trivial CNM message parser using batch parsing is shown in listing 3.3.

```
1 def unescape(s):
2     for k, v in {'\\0':'\0', '\\n':'\n', '\\_':' ', '\\-':'-',
      '\\\\':'\\'}.items():
3         s = s.replace(k, v)
4     return s
5 header = conn.readline()
6 version, intent, *params = header.rstrip('\n').split(' ')
7 intent = unescape(intent)
8 params = {unescape(k): unescape(v) for k, v in (p.split('=') for p in
      params)}
9 body = conn.read()
```

Listing 3.3: Trivial CNP parser in Python

Header composition is also very simple and fast: each field is written into the stream with special characters in values escaped (which can be done

byte-by-byte without having to maintain any state) and then either a space or a line feed (for the last field) is written after it.

3.4 Selectors

A feature that CNP introduces are the content selectors. Selectors facilitate content selection and filtering within the requested resource. A selector query can be provided in a request as a parameter and, if the server supports that selector on the requested resource, the response's contents are the results of applying that selector.

The specifications currently defines three selectors:

- `byte`: Select a subset of the requested content using a byte range.
- `info`: Get the response header that would be used for a request to the specified resource without having to retrieve the resource's contents.
- `cnm`: Retrieve a subset of a CNM document based on semantic selector queries; see Section 4.5 for a detailed overview.

For example, the `byte` selector can be used similarly to HTTP's `Range: bytes` header to download a byte slice of the requested resource. This can be useful for various cases, including resuming interrupted downloads, concurrent loading of a single resource and seeking video content. Example request/response pairs without selectors and with a `byte` selector are visible in listing 3.4.

```
1 cnp/0.4 /files/hello.txt
2 cnp/0.4 ok length=14
3 Hello, world!
4
5 cnp/0.4 /files/hello.txt select=byte:-5
6 cnp/0.4 ok length=11 select=byte:0-5
7 Hello
```

Listing 3.4: CNP request/response with byte content selector

More selectors can also be introduced later. For example, a `time` selector could be used to retrieve a timespan from e.g. audio or video content without relying on its format supporting byte-based seeking that is commonly used for HTML5 video in WebM or MP4 containers nowadays.

3.5 Comparison with HTTP

Most CNP features are modeled on their equivalents in HTTP. However, the main goals for CNP is to be simple, consistent and well-defined. Thus, CNP has less features than HTTP and the ones that both share may be slightly different in CNP. CNP focuses on the features that are important for content delivery, primarily meant to be used for retrieval of CNM documents and media content embedded in them.

3.5.1 Syntax and parsing

The most obvious difference between CNP and HTTP is the syntax. CNP's header is a single line of lowercase text, while HTTP headers consist of multiple lines of case-insensitive parameters and end with a blank line.

HTTP headers resemble SMTP (email) headers. While the general syntax resembles `Parameter-Key: parameter value`, parsing them is more complex than merely splitting the string on the first occurrence of a colon character, since the specification allows inserting arbitrary whitespace and empty elements in between individual elements of HTTP header values [3]. Additionally, HTTP header values use the ISO-8859-1 text encoding by default, which is often overlooked by implementations (usually either by restricting it to ASCII or by using UTF-8). HTTP header parameter keys are also case-insensitive, which means that they usually need to be normalized before use.

On the other hand, the CNP header is a simple string of space-separated values that ends with the first line feed byte. The CNP header is defined as a bytestring, though all predefined values consist of ASCII characters. To parse it, splitting by space and then splitting all parameters by the equals sign is

sufficient. Arbitrary binary values may be used as both parameter keys and values, with only a few specific bytes needing to be escaped into a specific two-byte escape sequence. Because the protocol header is a case-sensitive bytestring and a specific string can only be escaped into exactly one bytestring, parameter key or value matching can be performed without needing to escape the value first by simply comparing it with a previously escaped search string.

Implementations of HTTP also perform value escaping in vastly different way; for example, the `filename` parameter of the `Content-Disposition` is treated differently in common Web browsers to the point where there is no interoperable way to encode non-ASCII filenames in it [26]. In CNP, all header keys and values, as well as the message intent string, are escaped with the same algorithm, removing any confusion about the proper way to insert a non-ASCII string in any of them.

All lines in HTTP headers end with a carriage return and line feed pair for historic reasons. The CNP header only ends with a line feed, which is often easier to parse, since only a single byte has to be matched and dealing with exceptions, such as either carriage return or line feed nor being present, unnecessary.

Between the standard value escaping method defined before and only a single line feed being used to delimit the header, response splitting attacks are less likely to happen on CNP. Despite the HTTP standard requiring a carriage return and line feed sequence (with the carriage return being optional) as the line delimiter, many HTTP servers will accept either of them on their own too. Improper input sanitization (e.g. only sanitizing the line feed or only handling both a carriage return and line feed sequence) in the headers can lead to response splitting, where misplaced carriage returns or line feeds in header (e.g. `Cookie` or `Referer`) values can lead to the attacker being able to insert custom headers and even end the sequence of headers and insert body data into the response. In CNP, that is significantly less likely to happen because there is only a single valid header delimiter byte (line feed) and implementations can and should use the same escaping function for all header

fields, regardless of which parameter it is.

3.5.2 Parameters

HTTP defines a myriad different header parameters [15, 3], with any single implementation unlikely to support them all. While the abundance of headers is the direct result of various features that certain implementations support, it also means that clients and servers have to perform capability negotiation or graceful fallback upon encountering an unsupported feature, adding additional complexity to implementations. CNP, on the other hand, only specifies a small amount of different header parameters, each with a distinct purpose directly related to content delivery. Parameters with similar functionality or few use cases are avoided, since implementations are more likely to end up not supporting them. Additionally, all defined parameters have strict expectations for their values, so clients and servers are less likely to implement them incorrectly, as such requests will end up rejected by correct servers.

Notably, the HTTP `Host` parameter has been merged with the request path into the request intent. Exactly one `Host` header is mandatory in every HTTP/1.1 request, so it makes no sense to keep it as a header (most of which tend to be optional). With it being a part of the request intent in CNP, implementations have to include it on every request (even if it is blank).

As the previous section mentioned, all CNP parameters have well-defined encoding used to escape reserved values. The same escape sequences are used in all parts of the message header regardless of their meaning, so unlike in HTTP, any value can be easily and correctly escaped in any place.

3.5.3 Response types

While HTTP represents response status primarily with numerical codes [15] roughly grouped into categories based on their general meaning, CNP uses a small amount of explicit intent strings, each representing exactly one different type of response.

Errors

CNP `error` response intent covers all HTTP `4XX` and `5XX` status codes. The error meaning is provided in the reason response parameter.

In most cases (interactive Web browser), all HTTP error responses are handled by simply showing the attached page and possibly displaying the error reason. CNP simplifies that by having one error response intent and fewer defined error reason values, which should cover most situations.

A notable case is the HTTP `400 Bad Request` error. While it was originally intended to be sent as a response to invalid HTTP syntax sent by the client [14, 15], it is often (mis)used to signify an application-level error, such as not providing a required `GET` parameter. CNP defines a specific reason (`reason=rejected`) for that to avoid confusing it with syntax errors (`reason=syntax`).

Redirect

HTTP provides several different `3XX` status codes for redirects [15], but not all `3XX` codes are redirects. This means that the client must know every new redirect code in order for redirection to work. The main difference between these redirects is whether they should be cached and whether non-idempotent requests, such as `POST`, should be allowed to be redirected.

CNP only has one redirect response intent. If an equivalent to the `Cache-Control` HTTP header is implemented in a future version of CNP, it may be used to allow redirect caching; otherwise, redirects have to be followed every time. In general, this usually means one extra round-trip when the user follows a hyperlink that gets redirected. CNP redirect must always be a blank (body-less) request, equivalent to a HTTP `GET`. Since these requests often contain data of potentially significant size, uploading it for every redirect would be wasteful, in addition to unsafe since every hop would receive the entire request body. If redirecting requests with body data is supported by a future version of CNP, it can be done using a response parameter. Modifying the intent itself is unnecessary, since it is still a redirect.

Chapter 4

Light-weight markup format

The *ContNet Markup*, CNM, is a lightweight markup language primarily meant to be used as the hypertext document markup format for the ContNet platform. It is a line-based Unicode structured text markup document format. It uses indentation-delimited blocks to define the document structure and inline rich text formatting for the actual text content.

The primary goals of CNM are simple parsing and composition by computers as well as being readable and writable by humans, while being resistant to common errors, such as incorrect closing tag order or elements having unsupported elements as children, in order to prevent implementations having to use a "quirks mode" [29] and making these errors a valid syntax.

The complete document markup format specification is available in Appendix B.

4.1 Syntax

A CNM document uses UTF-8 as the character encoding. UTF-8 can encode all Unicode characters, is ASCII-compatible and has various other beneficial features [10]. It is the most popular character encoding, currently used by about 90% of all websites [28]. This means that document parsers do not have to support a large variety of encodings and also makes it less likely of

documents using an incorrect character encoding.

The primary syntax of a CNM document consists of nested blocks defined by their indentation. These blocks define the semantic structure of the document, such as sections, lists of items and tables. Each block begins with a line containing the block name and optional block arguments, followed by an arbitrary number of lines indented with one horizontal tab character representing the block body. An example of a document block structure is visible in listing 4.1.

The block body can contain either more nested blocks or text content, depending on the block. Blocks that contain other blocks are used to build the document structure and introduce semantic elements that contain other contents, such as sections, lists and tables. Blocks that contain text are leaf blocks in the document block tree and contain the actual contents. These contents can be, for example simple plain text, formatted text containing inline semantic markup or embedded external content such as images.

```
1  title
2      Hello, world!
3  content
4      section Lorem ipsum
5          text
6              Lorem ipsum dolor sit amet, consectetur adipiscing elit,
7              sed do eiusmod tempor incididunt ut labore et dolore
8              magna aliqua.
9          section Ut enim ad minim veniam
10             text
11                 Duis aute irure dolor in reprehenderit in voluptate
12                 velit esse cillum dolore eu fugiat nulla pariatur.
13     section Excepteur sint occaecat
14         text
15             Cupidatat non proident, sunt in culpa qui officia
16             deserunt mollit anim id est laborum.
```

Listing 4.1: CNM document with indentation-defined blocks

All block arguments and the body of various blocks contain simple text.

That is text that has any whitespace characters collapsed into single spaces and escape sequences resolved. Since whitespace is collapsed, simple text can be written over several lines for readability and still ends up as a single line. In the document contents, the plain text block can use an empty line to represent a paragraph break, so paragraphs can be used despite simple text replacing line feeds with spaces.

The simple text escape sequences are similar to the common C-style escape sequences consisting of a backslash character and a sequence representing the escaped character. Notable ones are:

- `\n`: Inserts a line feed (not a paragraph) that will not be replaced with a space.
- `\␣`: Inserts a raw space; can be used to write several sequential spaces without them being collapsed into a single space.
- `\x##`, `\u####` and `\U########`: With hexadecimal characters in place of `#`, inserts the specified Unicode codepoint; can be used to insert Unicode characters without having to find and type or paste the raw character.

The formatted text block contains more advanced inline text formatting and additional escape sequences, which are described in Section 4.3.

4.2 Structure

The top-level of the document contains blocks that define page metadata or contain the page contents. There are currently four defined top-level blocks:

- `title`: Contains the document title in the block body.
- `links`: Contains lines of (optionally) named hyperlinks to arbitrary URLs; can be used to add links to relevant pages on the website or other websites.
- `site`: Contains a sitemap; can be used to provide a hierarchical sitemap of the current website for easier navigation.
- `content`: Contains the page contents.

The `title`, `links` and `site` blocks represent metadata, as shown in listing 4.2. They do not contain the primary page contents, but will still be visible on the rendered page.

```
1  title
2      This is the Page Title
3  links
4      cnp://example.com/ Link to example.com
5      cnp://contnet.org/ Link to the ContNet website
6  site
7      path Link to /path
8          to Link to /path/to
9              folder Link to /path/to/folder
10         other_path This is a link to /path/other_path
```

Listing 4.2: CNM document with top-level metadata blocks

The `content` block contains the actual page content. It can contain arbitrary content blocks, which are visible in the main section of the page. Specific content blocks are described in Section 4.2.1.

Multiple instances of the same top-level blocks are merged together. Due to that, **content can be easily added into the document without modifying the current data by simply appending it on the end**. Additionally, since top-level block names never have indentation, a server can look for the presence of the `site` or `links` block using a simple string matching expression and, if one was not found in the document once it is fully served, append it to the output stream without having to fully parse and recompose the document or scan for these blocks before starting serving.

4.2.1 Content blocks

The `content` top-level block contains the entire body of the document. All of `content`'s child blocks represent the document content.

section

The `section` block represents a division of the contents with an optional title. With a block argument containing the section title, the block represents a section in the document. If nested inside other `section` blocks, it can represent a subsection of the parent section. The section block can contain any other content blocks, which count as semantically belonging to this section.

list

The `list` block represents an unordered or ordered list. The block argument may be `ordered` or `unordered` to choose the list type; when absent, the latter is assumed.

The contents of the `list` block can be arbitrary content blocks. Each direct child block represents one list item. A `section` block without a title can be used to group multiple blocks into a single item.

table

The `table` block can be used to construct two-dimensional tables. The contents of the block may be one of two special blocks: `header` or `row`. Both of these blocks act the same: they represent one line in the table (with `header` being an emphasized header line for the following rows) and contain other blocks, each of their child blocks representing one table cell.

embed

The `embed` block embeds external resources into the document. It takes a file type and a URL as the block arguments and an optional description text as the block body. It can be used to embed images, videos and various other resources into a document. If the user-agent cannot display the linked resource type in the page, it may simply provide a hyperlink that allows the user to access the resource directly.

raw

The `raw` block can be used to insert arbitrary text into the document. The contents of the block are not parsed, excluding removing the leading tab characters that define the block body. That way, source code and other text that might contain characters that would conflict with CNM text escape sequences can be safely inserted into a document.

Additionally, the `raw` block can be given a file type or syntax name as the argument. That may be used by user-agents to perform syntax highlighting on the block contents when it contains code in a supported syntax.

text

The `text` block is the primary way to insert text content into the document. The block body contains the text, while the block argument defines what kind of text it contains. Currently, three text formats are defined: `plain`, `pre` and `fmt`. If no argument is provided, the `plain` format is assumed.

A `text plain` block contains CNM simple text, where whitespace is collapsed and escape sequences resolved. Additionally, empty lines are used to delimit paragraphs, so multiple paragraphs can be specified in a single `text plain` block instead of requiring multiple blocks.

A `text pre` block is parsed similarly to a `raw` block, except that escape sequences are still resolved. Unlike `text plain`, this block does not support paragraphs and whitespace is not collapsed.

`text fmt` contains text with semantic markup formatting. This format is described in more detail in Section 4.3.

4.3 Formatted text

CNM supports inline text formatting in `text fmt` content blocks. This formatting defines semantic meaning for the formatted text, which is then usually rendered using styled text.

The formatted text paragraph is parsed the same way as a simple text paragraph, except that formatting toggles and additional escape sequences are resolved. For each toggle character, an escape sequence using that character prefixed with a backslash is defined and produces a raw character that does not act as a part of a toggle.

A format is toggled using two specific consecutive characters. Each toggle either enables the format if it is not currently active or disables it otherwise. The toggles for different formats do not have to be used in LIFO order. A paragraph of text starts with all formats disabled and all formats implicitly end with the end of a paragraph.

Formatted text current supports the following formats:

- `**`**Emphasized**: Toggle is two asterisks. Represents text emphasis. It is meant to be used to denote emphasized or stressed parts of text and is usually rendered in a bold font.
- `__`*Alternate*`__`: Toggle is two underscores. Indicates text in an alternate voice or mood that is offset from the normal text and is usually rendered in an italic font.
- ``` ``Code`` ```: Toggle is two grave accent characters. Represents computer code or similar text that is usually not a part of spoken language and is usually rendered in a monospaced font.
- `""`“Quoted”`""`: Toggle is two quote marks. Represents inline quotations.
- `@@`Hyperlink`@@`: Toggle is two at signs. The format takes an argument as the first non-whitespace field inside it, which represents the URL that the hyperlink points to, while the rest of the text is the hyperlink text. For example, `@@cnp://example.com/ This is a link@@` is a link to “`cnp://exmaple.com/`” with the visible text being “`This is a link`”.

4.4 Parsing

The block structure of CNM is easily parsed line-by-line in one pass while only keeping a block stack as the state. Blocks in a block context (such as

the contents of any container block, e.g. `contents` or `section`) are parsed by reading the input document by lines. If a non-empty line has less indentation than the current size of the block stack, the current block ends and is popped off the block stack, while the parsing continues in the parent block. Otherwise, if the line is not empty, the child block name and the block's arguments are parsed by splitting the line by whitespace, then pushing that block onto the stack and, if it is a container block, repeating this procedure for it.

Non-container blocks, such as `text`, are parsed in a similar way, except that instead of parsing child block names, the contents of the block are parsed as text. When parsing a text block that does not preserve whitespace, each line of the contents is read, its whitespace collapsed and escape sequences resolved, then joined to the previous line (if any) with a space to collapse line feeds. In case of the `text plain` block, reading an empty line concludes the last paragraph.

The `text fmt` block is parsed just like the `text plain` block, except that additional escape sequences are resolved (to permit using formatting toggles as characters) and formats are applied for the current paragraph each time a formatting toggle is encountered. Since the toggles are not ambiguous or context-dependent, the text can simply be scanned for non-escaped character sequences representing the toggles, then removing them and marking the following span of text with the specified format.

In general, parsing is unambiguous and requires only one pass through the document, no lookahead and minimal state. The document can be parsed from a stream without storing the entire parsed document in memory, as long as parsed contents are immediately processed.

4.5 Selectors

CNM selectors are query strings that identify specific sections in a CNM document. They can be used to select a section in the document for the purpose of moving it into view or to filter a document to select only certain

sections and their content. The primary uses of CNM selectors are `select=cnm:` selectors in CNP, as described in Section 3.4, and in page anchors in the hash fragment section of URLs to scroll the page to the position of the section.

Combined with CNP selectors, CNM selectors can be used to load a large document in semantic layers. First, a request for the top-level of the document is made, omitting the contents of all sections in the document contents. When the user wants to read a section, the user-agent can expand it with a new request using selectors, retrieving its contents without subsection contents. That way, very large documents can be loaded in steps, retrieving only the part of the content that the user is interested in, saving bandwidth and making the initial page loading time shorter.

CNM selectors come in three variants:

- **Title selector:** selects the first section with a specific title anywhere in the document. This selector cannot be used to select a section with the same title as a section above it in the document.
  Example: `#Section Title`
- **Title path selector:** selects a section by hierarchically traversing sections in the document, selecting the first section with the title corresponding to its path segment on each step, starting at the root of the `content` top-level block. This selector cannot be used to select a section with the same title as a section above it in the current context.
  Example: `/Top-level Section/Subsection 3/Section Title`
- **Index path selector:** selects a section by providing a list of section indices, corresponding to their position among other sections in the parent section (or the `content` block, in case of top-level sections). This selector can be used to select arbitrary sections in the document.
  Example: `$1.3.2`.

The selectors select sections using implicit semantic identifiers. This means that they can be used to select any sections in the document without requiring the writer to annotate them with identifier metadata and the meaning of

the selectors is semantic and clear even to humans without having to look through the document source markup.

```
1  title
2      Document Title
3  content
4      text
5          This is on the top level
6      section Top-level Section
7          section Subsection 1
8              text
9                  Text in first subsection
10         section Subsection 2
11             text
12                 Text in second subsection
13         text
14             Text in a top-level section
15         section Subsection 3
16             text
17                 Text in third subsection
18             section Section Name
19                 text
20                     Text in a sub-subsection
21     section Another top-level section
22         text
23             Text in another section
```

Listing 4.3: Example CNM document

When filtering a document rather than just identifying a section in it, the selectors select the targeted section with all of its content and all parent blocks up to the root `content` block, but excluding any sibling blocks of the targeted block or its parent blocks. Additionally, a "shallow" content selector can be used by prefixing a selector with an exclamation mark. Shallow selectors filter the content similarly to normal selectors, but they exclude the contents of any child sections of the target section, leaving any other blocks and child section titles. For example, using the selector `!/Top-level Section/Subsection 3` on

the document provided in listing 4.3 results in the document in listing 4.4.

```
content
    section Top-level Section
        section Subsection 3
            text
                Text in third subsection
            section Section Name
```

Listing 4.4: Result of a shallow selector on Subsection 3

As a special case, an empty selector selects the entire document, including non-`content` top-level blocks. If the shallow selector modifier is applied to it, then the content of sections within the `content` block is omitted, but the rest of the document metadata is kept intact, as shown on listing 4.5. This way, user-agents can request the empty shallow selector to get the top-level of document content and display collapsed sections, then use shallow section selectors to retrieve contents of sections when the user clicks on them.

```
title
    Document Title
content
    text
        This is on the top level
    section Top-level Section
    section Another top-level section
```

Listing 4.5: Result of the empty shallow selector `!` on document from listing 4.3

4.6 Comparison with HTML

HTML supports various formatting options, including all supported in CNM. However, this comes at the cost of more complex parsers and mixing content with layout.

A subset of HTML only using semantic HTML5 elements is relatively similar to CNM in feature scope, defining more features and semantic format-

ting. However, the cost of that is both a more complicated implementation and more user confusion. For example, users may confuse the `<b>` tag (representing stylistically different but not more important text) with `<strong>` tag (gives text strong importance), `<em>` tag (text with stress emphasis) and even `<i>` (non-emphasized text with a different semantic meaning), all of which are somewhat similar but mean different things. Additionally, due to their meaning in earlier versions of HTML [18], users might just use `<b>` and `<i>` simply when they want, respectively, bold or italic text, rather than paying any attention to their semantic meanings. CNM defines the text formats (especially `**emphasized**` and `__alternate__` being roughly semantically equivalent to `<em>` and `<i>`) as semantic markup directly, while not defining multiple versions of semantically similar text markup that would by default be rendered the same on common implementations

```
<!doctype html>
<html><head><meta charset="utf-8"><title>Hello,
world!</title></head><body><h1>Hello, world!</h1><section
id="Lorem ipsum"><h2>Lorem ipsum</h2><p>Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua.</p><section
id="Ut enim ad minim veniam"><h3>Ut enim ad minim veniam</h3><p>Duis
aute irure dolor in reprehenderit in voluptate velit esse cillum
dolore eu fugiat nulla pariatur.</p></section></section><section
id="Excepteur sint occaecat"><h2>Excepteur sint occaecat</h2><p>
Cupidatat non proident, sunt in culpa qui officia deserunt mollit
anim id est laborum.</p></section></body></html>
```

Listing 4.6: HTML version of the document from listing 4.1

A notable difference between CNM and HTML is the syntax. CNM uses indentation-defined text blocks, while HTML uses SGML-inspired syntax containing opening and closing tags contained in angle brackets. This gives CNM more readable syntax by default, as shown in listing 4.1 compared to the equivalent HTML document in listing 4.6. It also separates structural syntax from the content (with inline text markup), since structure is implied

by the indentation.

Another notable feature of CNM is that metadata, such as selectors, is implicit. Unlike HTML, where element IDs have to be explicitly defined (as shown in listing 4.6), CNM sections implicitly use their titles and positions within the document as selectors. Other metadata, such as tables of contents, can also be generated without having to be specified in the document source. That way, all contents of the `content` block are purely semantic.

4.7 Comparison with Markdown

The inline text markup used by CNM is close to other lightweight markup languages, the most popular of which is Markdown. The main differences between CNM and Markdown are that CNM is more structured, more precisely defined, less ambiguous, more extensible, more semantic, easier to parse and not bound to HTML.

Markdown was initially defined by a reference implementation [21] and has no standard specification, which has resulted in the existence of various different flavors of it [9, 17]. While there have been efforts to standardize one common version of Markdown [9], many incompatible implementations and documents still exist. Markdown also contains various ambiguities due to being defined by imprecise documentation and a reference implementation. CNM avoids these problems by being precisely defined with a specification from the beginning.

Several features of Markdown require parsing in multiple passes, such as reference links. Other parts require significant lookahead while parsing. CNM can be parsed in one pass one token at a time by only keeping a minimal state containing the stack of blocks for the current line.

CNM has a block-based structure where most of the document structure is defined via indented blocks, with only text markup being inline. The block structure is easy to extend by introducing new blocks and easier to parse because of explicit scope. In Markdown, sections are only delimited by the

start of the next section with no way to explicitly end the current section. It is also nearly impossible to extend due to the lack of generic parts of syntax [17], since almost all syntax is composed from dedicated symbol characters.

Finally, Markdown allows inline HTML and is defined by the HTML that should be generated from it, making it harder to adapt for documents that will not result in HTML. CNM is a standalone document format and while it can be converted to HTML, it does not rely on any feature in other markups.

For a comparison of implementations, a common Go library that parses Markdown and converts it into HTML, *blackfriday* [27], 4000 lines of Go code in version `2.0.0`. *cnm-go*, which can also compose CNM documents alongside parsing them, contains around 2000 lines of Go code in version `0.4.0`, while version `0.2.1` of the *cn-http* application contains around 400 lines of Go code and 200 lines of HTML markup that facilitates translating parsed CNM documents into HTML.

Chapter 5

Implementation

The initial implementation of the specifications is a pair of libraries in the Go programming language. The libraries implement parsing and composition for both the protocol and the document format, as well as a simple client and server for the protocol. These libraries are used in a server and client application.

5.1 Libraries

The core implementation of the specifications consists of two libraries for the Go programming language: *cnp-go*, implementing the protocol, and *cnm-go*, implementing the markup language.

5.1.1 cnp-go

The *cnp-go* library implements all features of the ContNet protocol. It can parse CNP messages into Go datatypes, such as strings and maps, and compose these into CNP messages. CNP header escaping and unescaping is also supported. It also implements a basic CNP client and server.

The primary interface to CNP messages are the `cnp.Request` and `cnp.Response` objects, both of which extend the `cnp.Message` type. The latter contains a CNP header, represented as the version number, an intent string and a

map of parameters, and the message body, if any. By using the function `cnp.ParseRequest` or `cnp.ParseResponse`, the client or server can parse, respectively, requests or responses from a connection. The request and response objects also have a method `.Write`, which can serialize and write it to a connection.

Both request and response objects have methods that allow simple access to standard header parameters, while also validating them at the same time and substituting the default value if the parameter is invalid. Additionally, validation of the message intent and all standard parameters can be performed to reject any invalid messages.

```
1  package main
2  import (
3      "io"
4      "os"
5      "contnet.org/lib/cnp-go"
6  )
7  func main() {
8      resp, err := cnp.Get("cnp://localhost/")
9      // generates the following request:
10     // cnp/0.4 localhost/
11     if err != nil {
12         panic(err)
13     }
14     io.Copy(os.Stdout, resp.Body)
15 }
```

Listing 5.1: CNP client retrieving a page from localhost using *cnp-go*

The client implementation provides the function `cnp.Get`, shown in listing 5.1, allows simple retrieval of CNP resources based on a URL. For more complex requests, the function `cnp.Send`, which takes a `cnp.Request` object as the parameter and is also used by `cnp.Get` internally, can be used. The client establishes a TCP socket using Go's `net.Dial` function, serializes a `cnp.Request` message into it, then parses a `cnp.Response` from the connection, while also

returning any errors that happened during the transaction.

The server is implemented with the `cnp.Server` object. It can be assigned a network address or hostname to listen on and a request handler, which is used to handle all incoming requests. Additionally, it supports access and error logging, which can be further customized with user-specified callback functions that can replace the default NCSA common log format [19] logger. Automatic request validation, which rejects any requests with invalid values of known parameters, can also be toggled.

The server listens on TCP on the specified address and port, waiting to accept incoming connection. As each connection is accepted, the handler is called in a coroutine, using Go's native concurrency support to handle a large amount of simultaneous connections without having to adjust the handler function.

An example of server usage is shown in listing 5.2.

```
1  package main
2  import (
3      "fmt"
4      "contnet.org/lib/cnp-go"
5  )
6  func main() {
7      cnp.ListenAndServe("localhost",
8          cnp.HandlerFunc(func(w cnp.ResponseWriter, r *cnp.Request) {
9              fmt.Fprintln(w, "Hello, world!")
10             // generates the following response:
11             // cnp/0.4 ok
12             // Hello, world!
13         }),
14     )
15 }
```

Listing 5.2: CNP server serving "Hello, world!" using cnp-go

While the client and server implementations could likely be improved and optimized, they are meant to be able to be used directly in Internet-facing

applications. Currently, rate-limiting and timeouts are not supported, relying on the user to handle that in their application manually.

5.1.2 cnm-go

The *cnm-go* library provides the support for parsing and composition of CNM documents, as well as an implementation of CNM selectors that can be used to find sections or filter content in parsed documents.

The function `ParseDocument` can parse a CNM document from an input stream, returning a `cnm.Document` object representing the contents of the document using Go structures defined in the library. As shown in listing 5.3, *cnm-go* can also compose new documents from these types programmatically.

The `cnm.Document` object contains the top-level blocks (`title`, `links`, `site` and `content`) as field values. The contents of the `content` block are represented as a tree of nested container blocks (such as `cnm.SectionBlock`, `cnm.TableBlock` and similar) and terminal blocks (`cnm.TextBlock`, `cnm.RawBlock` and `cnm.EmbedBlock`).

The `cnm.TextBlock` type, representing a `text` content block, additionally holds one of several different text content types:

- `cnm.TextPlainContents`: a list of strings representing parsed text paragraphs from a `text plain` block.
- `cnm.TextPreContents`: a single string representing parsed text contents from a `text pre` block.
- `cnm.TextRawContents`: a single string representing unparsed text contents from a `text` block with an unknown format; this type is also used by the `cnm.RawBlock` type.
- `cnmfmt.TextFmtContents`: a list of paragraphs, with each paragraph being represented as a list of `cnmfmt.Span` types, each of which contains a span of parsed formatted text and the state of all text formats for that span of text.

The `cnm.Document` type also has the `.SelectBlock` function, which finds a

section (as the type `cnm.SectionBlock` using a CNM selector and the `.Select` function, which takes a CNM selector query as a parameter and returns a new `cnm.Document` containing contents filtered through the given content selector.

```go
package main

import (
    "os"
    "contnet.org/lib/cnm-go"
)

func main() {
    doc := cnm.NewDocument()
    doc.Title = "Hello, world!"
    doc.Content = cnm.NewContentBlock("content")

    sec1 := cnm.NewSectionBlock("Lorem ipsum")
    text1 := cnm.NewTextPlainBlock([]string{
        "Lorem ipsum dolor sit amet, consectetur adipiscing elit, " +
            "sed do eiusmod tempor incididunt ut labore et dolore " +
            "magna aliqua.",
    })
    sec1.AppendChild(text1)
    sec2 := cnm.NewSectionBlock("Ut enim ad minim veniam")
    text2 := cnm.NewTextPlainBlock([]string{
        "Duis aute irure dolor in reprehenderit in voluptate " +
            "velit esse cillum dolore eu fugiat nulla pariatur.",
    })
    sec2.AppendChild(text2)
    sec1.AppendChild(sec2)
    doc.Content.AppendChild(sec1)
    sec3 := cnm.NewSectionBlock("Excepteur sint occaecat")
    text3 := cnm.NewTextPlainBlock([]string{
        "Cupidatat non proident, sunt in culpa qui officia " +
            "deserunt mollit anim id est laborum.",
    })
    sec3.AppendChild(text3)
    doc.Content.AppendChild(sec3)

    doc.Write(os.Stdout)
}
```

Listing 5.3: Recreating the document from listing 4.1 using cnm-go

5.2 Server

The initial implementation of a server application is a simple server called *cn-fileserver*. It simply serves files from a specified directory over CNP.

The server uses the *cnp-go* library to create a CNP server. For every request received, it tries to serve a file based on the requested path. If that file is a directory, *cn-fileserver* lists the its contents in the form of a CNM document generated via the *cnm-go* library.

File metadata, such as filename, size and modification time, is attached automatically. If the client makes a request with an `if_modified` header parameter and the file has not been modified since then, the server responds with a `not_modified` response, which indicates that the client should use the cached copy.

Additionally, *cn-fileserver* has support for the `info`, `byte` and `cnm` CNP selectors. The first two are implemented directly in *cn-fileserver*, while the `cnm` selector uses the selector functionality of the *cnm-go* library.

The server uses just a few megabytes of memory for normal operation (around 5 MiB while idle, going up to around 20 MiB on a thousand concurrent connections) and can serve all requests concurrently, thanks to *cnp-go* and the Go language's support for lightweight concurrency.

5.3 HTTP gateway

The *cn-http* application is a HTTP server that translates HTTP requests to CNP requests, sends these to a specified CNP upstream server and translates the responses back into HTTP. If the CNP response contains a CNM document, that document is translated into HTML.

cn-http's primary use is to act as a gateway from the Web to ContNet servers. It enables access to ContNet content using standard Web browsers. Since most current Internet users do not use ContNet clients, this gateway allows ContNet content to be exposed to them too.

Additionally, *cn-http* can act as a ContNet browser, where instead of

using a static upstream server, it provides the user with an address box that navigates to ContNet pages based on the URL entered into it.

The *cn-http* application supports all features of CNM. The contents of the `content` block of a CNM document are translated into semantic HTML5 elements. The HTML output can be styled with CSS and made interactive using JavaScript. The default stylesheet, shown in figure 5.1, uses a very simple black-on-white style, but could be replaced with a more intricate one if a more complex design is desirable.

Figure 5.1: The CNM document from listing 4.1 rendered with *cn-http*

cn-http supports CNM selectors in the hash fragment of the URL by scrolling to the specified section; the default JavaScript code will also collapse all but the named section. Additionally, a table of contents that uses these selectors is automatically generated based on sections in the content. Limited support for CNP `select=cnm:` selectors is also available via the `?select=` URL query parameter, but is currently not used in the implementation. In a future

version, it could be used to dynamically load section contents when the user expands them.

Chapter 6

Conclusions and further work

6.1 Conclusions

This thesis specifies a content transfer protocol and a document format that together form a lightweight content delivery platform with a focus on simplicity, correctness and ease of implementation. This platform is also implemented in the form of a pair of libraries that handle the protocol and the document format, a file server and a gateway between the new content delivery platform and the Web that can also be used as a browser for the new platform.

The new content delivery platform can be used as an alternative to the part of the Web that serves content-focused static documents. It is significantly simpler to implement and use. Documents written in the newly specified document format only define semantic formatting, leaving presentation to the user-agent, which allows the documents to be correctly displayed regardless of the user-agent device type, screen size and other attributes.

The average number of objects (stylesheets, scripts, images and similar) embedded in HTML pages has increased by 260% between 2003 and 2010 [8], with the average page size getting about five times as large. To load these objects concurrently, Web browsers usually use several simultaneous TCP connections. That, however, breaks some assumptions that TCP was built

upon and negates TCP congestion control, hurting network performance. Users in low bandwidth environments will be affected the most, possibly ending up in a congestive collapse [8]. Since CNM documents do not include stylesheets or scripts, the only external content that has to be loaded after the page consists of embedded multimedia elements, such as images. Since these are not critical for rendering the page layout, they can be retrieved one by one on a single connection. Intelligent user-agents can even prioritize loading visible embedded content while pausing any that is currently not displayed.

Protocol-level semantic document content selectors enable implementation of browsers that load pages breadth-first. That way, even very large pages can have fast initial loading times, with the contents of sections and any embedded content within them only being loaded when the user wants to view that section. This way, each page load is smaller, does not include the content the user is not interested in and embedded content can be segmented into several section in a way where no section has too much external content despite a large amount of it present in the entire document. Unlike HTML, ContNet pages do not require any extra work or scripts on the part of the page designer to support that feature.

The combination of ease of implementation and low complexity means that the content delivery platform can be implemented for and used on specialized low-performance devices, such as embedded devices that are a part of the Internet of Things. These devices traditionally had restricted access to the Web, since the current Web technologies demand very complicated implementations that usually need enormous amounts of system resources far in excess of what embedded devices tend to have.

Alternatively, the document format can be used outside the content delivery platform as a source for Web pages. Since it is simple to read and write for humans, as well as containing only semantic markup, it can be used similarly to other existing lightweight markup languages, where it is translated into HTML on the server before being served to clients. That way, the document format can be useful even if the content delivery platform does

not see a widespread use.

Regardless of the above, the entire project can be used as a basis for a different take on an implementation of a part of the Web or at least an example of an attempt at sanitization of Web content delivery.

6.2 Further work

There are opportunities for extension of both the content transfer protocol and the document markup format.

The protocol could benefit from several optimizations:

- Batch requests: avoids having to make many round-trips when transferring embedded content.
- Form submission: in combination with forms in the document format, allows for data submission by users.
- Sessions: support for persistent state, hopefully in a saner way than HTTP cookies.
- Compression: support for optional compression of message body in order to save bandwidth and make transfers faster.
- Encryption: if TLS is used for all connections, the protocol is significantly safer against various attacks and better at preserving user privacy.

The primary missing feature of the document format are forms. Without them, creating even simple types of user content submissions, such as query input fields in order to run a search on the server, is not possible. In combination with an implementation in the protocol, this would allow user data submission in general, extending the platform from pure content delivery into a more general direction.

List of Figures

Listings

Bibliography

[1] Allied Business Intelligence, Inc. More Than 30 Billion Devices Will Wirelessly Connect to the Internet of Everything in 2020. https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne/. Accessed: 2017-09-12.

[2] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Albert. The Internet Gopher Protocol (a distributed document search and retrieval protocol). RFC 1436 (Informational), March 1993.

[3] James Antill. HTTP for servers. http://www.and.org/texts/server-http. Accessed: 2017-09-11.

[4] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015.

[5] Robin Berjon, Silvia Pfeiffer, Steve Faulkner, Erika Doyle Navara, Theresa O'Connor, Ian Hickson, and Travis Leithead. HTML5. W3C recommendation, W3C, October 2014. http://www.w3.org/TR/2014/REC-html5-20141028/.

[6] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[7] Bert Bos. Cascading style sheets level 2 revision 2 (CSS 2.2) specification. W3C working draft, W3C, April 2016. http://www.w3.org/TR/2016/WD-CSS22-20160412/.

[8] Jay Chen. *Re-architecting Web and Mobile Information Access for Emerging Regions*. PhD thesis, Courant Institute of Mathematical Sciences, September 2011.

[9] CommonMark team. CommonMark. `http://commonmark.org/`. Accessed: 2017-09-12.

[10] Russ Cox. UTF-8: Bits, Bytes, and Benefits. `https://research.swtch.com/utf8`. Accessed: 2017-09-06.

[11] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.

[12] Jon Ferraiolo. Scalable vector graphics (SVG) 1.0 specification. W3C recommendation, W3C, September 2001. http://www.w3.org/TR/2001/REC-SVG-20010904.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (Proposed Standard), January 1997. Obsoleted by RFC 2616.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[15] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231 (Proposed Standard), June 2014.

[16] Gartner. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. `http://www.gartner.com/newsroom/id/3598917`. Accessed: 2017-09-12.

[17] Eric Holscher. Why You Shouldn't Use "Markdown" for Documentation. `http://ericholscher.com/blog/2016/mar/15/dont-use-markdown-for-technical-docs/`. Accessed: 2017-09-12.

[18] Arnaud Le Hors, Dave Raggett, and Ian Jacobs. HTML 4.01 specification. W3C recommendation, W3C, December 1999. http://www.w3.org/TR/1999/REC-html401-19991224.

[19] IBM. Log File Formats - NCSA Common (access log). `http://publib.boulder.ibm.com/tividd/td/ITWSA/ITWSA_info45/en_US/HTML/guide/c-logs.html#common`. Accessed: 2017-09-06.

[20] Patrick D F Ion, Robert R Miner, and David Carlisle. Mathematical markup language (MathML) version 3.0 2nd edition. W3C recommendation, W3C, April 2014. http://www.w3.org/TR/2014/REC-MathML3-20140410/.

[21] S. Leonard. The text/markdown Media Type. RFC 7763 (Informational), March 2016.

[22] Microsoft. Microsoft Edge requirements and language support. `https://docs.microsoft.com/en-us/microsoft-edge/deploy/hardware-and-software-requirements`. Accessed: 2017-09-11.

[23] Sangwhan Moon, Steve Faulkner, Travis Leithead, Arron Eicholz, and Alex Danilo. HTML 5.2. Candidate recommendation, W3C, August 2017. https://www.w3.org/TR/2017/CR-html52-20170808/.

[24] Mozilla. Firefox 55.0.3 System Requirements. `https://www.mozilla.org/en-US/firefox/55.0.3/system-requirements/`. Accessed: 2017-09-11.

[25] Opera Software. Opera 42.0 system requirements. `http://www.opera.com/download/requirements`. Accessed: 2017-09-11.

[26] Julian Reschke. Test Cases for HTTP Content-Disposition header field (RFC 6266) and the Encodings defined in RFCs 2047, 2231 and 5987. `http://greenbytes.de/tech/tc2231/`. Accessed: 2017-09-11.

[27] Russ Ross. Blackfriday: a markdown processor for Go. https://github.com/russross/blackfriday. Accessed: 2017-09-12.

[28] W3Techs. Historical trends in the usage of character encodings for websites, September 2017. https://w3techs.com/technologies/history_overview/character_encoding. Accessed: 2017-09-06.

[29] WHATWG. Quirks Mode Standard. https://quirks.spec.whatwg.org/. Accessed: 2017-09-11.

[30] Michal Zalewski. HTTP cookies, or how not to design protocols. http://lcamtuf.blogspot.com/2010/10/http-cookies-or-how-not-to-design.html. Accessed: 2017-09-12.

[31] Marko Čeferin. lib/cnm-go. https://contnet.org/lib/cnm-go/. Accessed: 2017-09-12.

[32] Marko Čeferin. lib/cnp-go. https://contnet.org/lib/cnp-go/. Accessed: 2017-09-12.

[33] Marko Čeferin. tool/cn-fileserver. https://contnet.org/tool/cn-fileserver/. Accessed: 2017-09-12.

[34] Marko Čeferin. tool/cn-http. https://contnet.org/tool/cn-http/. Accessed: 2017-09-12.

Appendices

Appendix A

ContNet Protocol specification, version 0.4 (2017-09-04)

A.1 Overview

CNP is a request-response application protocol meant to facilitate hypertext content requests and delivery over the Internet. It is designed to be an alternative to a part of HTTP; specifically, the subset dealing with static content (text, images, videos and similar). The protocol is designed to be simple, easy to implement and unambiguous, as well as strict in order to avoid having to support multiple incorrect implementations.

It is a binary protocol. While all of the defined keywords are also valid ASCII text (lowercase letters, numbers, symbols and whitespace), CNP does not use any character encoding. Rather, these keywords should be treated as specific bytestrings. Every mention of CNP strings in the specification refers to bytestrings whose content is the provided ASCII-encoded text. This is done to avoid problems with encoding and case insensitivity and simplify parsing, since each keyword has exactly one valid bytestring representation.

For now, the protocol implements only a limited amount of simple optimizations. While it would be possible to make it slightly faster by including more, that would also increase its complexity and increase the likelihood

of implementations supporting a very small or even incorrect subset of the specification. Additional simple optimizations may be added in future versions of CNP.

A.2 CNP message

CNP messages are composed of a header and an optional body. The header is delimited by a newline. Both request and response share the same syntax, with the differences being the semantic meaning of values.

A.2.1 Header

The header consists of the CNP version, message intent and an arbitrary number of parameters, using a space as the delimiter between fields.

The header field delimiter is a single space byte (`0x20`). Multiple successive spaces in the header are a syntax error. A space cannot appear as itself in any header value; it must be escaped into a specific byte sequence. This makes splitting a valid header by spaces safe and correct.

The CNP version part of the header is the string `cnp/` followed by a tuple of the major and minor version numbers without leading zeroes on non-zero versions, separated by a period (e.g. `cnp/0.4`). Even if the specification and implementation use a patch version, it is not provided in the message header and must have no impact on protocol compatibility. CNP implementations within the same major version should be generally compatible with other minor versions for basic requests, excluding some noncritical feature discrepancy; unknown parameters are ignored. An exception to this are all versions in the major version `0`; since it's the development version, the protocol may be completely rewritten between two minor versions.

The intent is a string that defines the type of the message. Valid values depend on whether the message is request or response.

Parameters are key-value pairs separated by an equals sign (`0x3d`). There may be any number of parameters in a message. The key and value can

be arbitrary bytestrings, including empty strings. The order of parameters can be arbitrary and implementations must not depend on it nor alter their behavior based on the order; sorting the parameters or shuffling their order produces an identical request. Each parameter key may appear at most once in a message; duplicate parameter key is an error. The equals sign must always be present for a parameter, even if the key or value is blank. A missing parameter is considered to have the blank value, so there is no difference between a parameter provided with a blank value and one not present in the header.

Header ends with a line feed byte (`0x0a`). Note that if there is a carriage return before it, it will be considered to be a part of the last token (parameter value or intent).

The message body is a blob of arbitrary binary data. There is no body delimiter or terminator; if the body ends with a newline, it is considered a part of the body data.

Example: `cnp/0.4 msg_intent param1=value1 param2=value2`

A.2.2 Escaping

The NUL (`0x00`), line feed (`0x0a`), space (`0x20`), equals sign (`0x3d`) and backslash (`0x5c`) bytes must be escaped by using a backslash and a specific character (see table below) in all parts of the message header, except when used to delimit header fields or parameter key and value. Using any of these characters when not otherwise specified by the syntax (such as space between intent parameters, equals sign between parameter key and value and line feed at the end of the header) is a syntax error. Other characters, such as tab and carriage return, do not have to be escaped and stand for themselves.

Each raw value can be escaped into exactly one escaped value and each escaped value maps 1:1 to raw values. Because of this, escaped values do not have to be unescaped or normalized before being handled, as long as they are compared with escaped versions of expected values.

The message body contains no escaping and is plain binary data (unless

specified otherwise by a parameter).

```
(NUL)   ->  \0
(LF)    ->  \n
(space) ->  \_
=       ->  \-
\       ->  \\
```

A.2.3 EBNF

```
raw_character = ? any byte except "\0", "\n", " ", "=", "\\" ? ;
escaped_character = "\\" ( "0" | "n" | "_" | "-" | "\\" ) ;
ident_character = raw_character | escaped_character ;
identifier = ident_character, { ident_character } ;
number = "0" | ( "1" ... "9" ), { "0" ... "9" } ;

version = "cnp/", number, ".", number ;
intent = identifier ;
parameter = [ identifier ], "=", [ identifier ] ;

header = version, " ", intent, { " ", parameter }, "\n" ;
body = { ? any byte ? } ;

message = header, body ;
```

A.3 Request

A CNP request is sent by a client that wants to request a resource from a server (or send some data to it).

A.3.1 Intent

In a request, the intent part of the header contains the hostname of the server concatenated with the path of the requested resource.

The hostname may be a domain name or an IP address, optionally with a port number, and it may not contain any slash (`0x2f`) bytes. In general, it should be the address that the client sends the request to. The format of the hostname is the same as in URLs.

The path is a Unix-style absolute filepath with an optional trailing slash. It may (and probably should) be cleaned up by CNP implementations before being processed by collapsing multiple consecutive slashes into single ones, removing path entries that consist of a single dot and resolving double-dot parent directories, while leaving a trailing slash if one was present (for example, the path `/../../.\/\/\/foo/bar/..` becomes `/foo` and `\/\/foo/bar/../` becomes `/foo/`). If possible, implementations should avoid sending filepaths that need to be cleaned. The minimal path is `/`; blank path is an error.

The hostname is case-insensitive, while the path is case-sensitive. Hostname may be normalized by clients and servers, but a specific path (after cleanup) represents one specific resource.

To separate the hostname from the filepath, split the intent before the first slash.

A.3.2 Parameters

The following request parameters are defined (default value is assumed if the parameter value is blank; any absent parameter must be treated as if it had the default value):

length

`length={number}`

The length of the request body data in bytes.

This field is required if any data will be sent to the server. If it's absent

or zero, the request must not contain any body data. Otherwise, the byte size of the request body data must be provided in this parameter.

Default value: `0` (empty body)

name

`name={identifier}`

The name of the content being sent in the request body. Must not contain slashes (`0x2f`) or NUL bytes (`0x00`), including escaped NUL bytes (`\0`).

Can be used to provide filename metadata. If the original filename was a Unicode filename instead of a bytestring, UTF-8 should be used to encode it.

Default value: empty (no name provided)

type

`type={identifier}/{identifier}`

MIME type of the content being sent in the request body.

Default value: `application/octet-stream`

if_modified

`if_modified={timestamp}`

Only send the resource in the response if it has been modified since the RFC 3339 UTC timestamp provided, according to the server's time. Otherwise, reply with a `not_modified` response and no body data.

The RFC 3339 timestamp format should be the following, as a strftime format string format: `%Y-%m-%dT%H:%M:%SZ`, for example: `1970-12-31T23:59:30Z`.

The timestamp should usually be either the server's `modified` parameter value from the time the cached resource was requested or the `time` parameter value from that request if the former was not provided. If neither was present in the previous response for the current request's intent, the client should not use an `if_modified` parameter.

An `if_modified` parameter may only be set on a request for a cached page when all of the host, path and selector are equivalent to the previously cached copy.

Default value: empty (do not perform this check, just send the resource)

select

`select={identifier}:{identifier}` `select={identifier}:`

Request the server to apply a filter on the contents of the requested path before sending it.

The parameter value consists of a selector name (that may not contain a colon), a colon byte (`0x3A`) and an optional selector query. The functionality of the selector depends on which selector is requested.

The following selectors are defined in this specification:

- byte `select=byte:{number}-{number}` `select=byte:{number}-` `select=byte:-{number}` `select=byte:-`

  The `byte` selector selects a subset of the content bytes. The `{from}` value represents the index of the first included byte; it defaults to `0`, the first byte in the contents, when absent. The `{to}` represents the index of the last included byte; it defaults to the last byte in the contents when absent. An end byte lesser than the start byte is invalid in a request. If the start byte index is greater than the content length, the response contents should be empty (`length=0`).

  If possible, implementations should support this selector.
- info `select=info:`

  The `info` selector selects the response header of a request to this path. It takes no query string; if one is provided, the selector is invalid.

  The response body (**not** header) should contain the CNP header line that would have been used to answer this request if it didn't contain the `info` selector. It should contain all parameters that an actual response header would, especially a potential `length` parameter.

  If possible, implementations should support this selector.

- cnm `select=cnm:{identifier}`

  The `cnm` selector selects content based on CNM content selector. See the CNM specification for more information.

Implementations may choose to support custom selectors. More selectors may be defined in the future.

If an unknown selector is requested, it should be ignored and the request responded to as if there was no selector.

If a known selector is requested for an unsupported path, the response should be an `error` with `reason=not_supported`.

If the selector query was invalid, the response should be an `error` with `reason=invalid`.

A.4 Response

CNP requests are answered by a CNP response by the server. Most of the time, the response will probably contain the requested resource.

A.4.1 Intent

The intent part of the response header is one of the defined response types. It signifies whether the response contains the resource requested, a redirect, report of an error or other results.

ok

The request was successfully resolved and the response body contains the requested resource.

not_modified

The resource has not been modified since the time in the request's `if_modified` parameter. The response body is blank and the client should use the cached resource.

redirect

The client should make a new request to the location provided in the `location` parameter. The response body may contain a page as if it was an `ok` response.

The new request should be blank (contain no body data, as if that path was just entered anew by the user).

Clients are not required to follow the redirect. Servers should not assume that a client will always follow the redirect immediately or at all. Interactive user agents may prompt the user for confirmation before opening the new page. Otherwise, following redirects up to a certain count is generally the expected behavior.

User agents do not have to display the provided page and may directly perform the redirect. If the redirect is not performed and a page was provided, it should be displayed instead.

error

There was an error answering the request. The server must also provide the `reason` response parameter. The body data may contain a page as if it was an `ok` response.

User agents do not have to display the provided page and may just inform the user of the error reason. If a page is provided, it should be used to inform the user of the details of the error.

A.4.2 Parameters

The following response parameters are defined (default value is assumed if the parameter value is blank; any absent parameter must be treated as if it had the default value):

length

`length={number}`

The length of the response body data in bytes.

If present, it must contain the length of the response body in bytes. The client should use that to delimit reading the response.

If this parameter is not provided or is blank, the client should read all data until the connection is closed, end of file is reached or equivalent. Despite that, the length parameter should be sent whenever possible. If the transport does not support an equivalent to an end-of-message signal, then the `length` parameter is required for responses.

A zero response `length`, which is not the default, means that the response has no body data.

Valid in all response types.

Default value: empty (read until EOF)

name

`name={identifier}`

The name of the content being sent in the response body. Must not contain slashes (`0x2f`) or NUL bytes (`0x00`), including escaped NUL bytes (`\0`).

Can be used to provide filename metadata. If the original filename was a Unicode filename instead of a bytestring, UTF-8 should be used to encode it.

Valid in response types where the body data can be file (`ok`, `redirect`, `error`).

Default value: empty (no name provided)

type

`type={identifier}/{identifier}`

MIME type of the content being sent in the response body.

Valid in response types where the body data can be file (`ok`, `redirect`, `error`).

Default value: `application/octet-stream`

time

`time={timestamp}`

The current time on the server as an RFC 3339 UTC timestamp.

May be sent with any response type, but most useful in the `ok` response, where it may be used by the client in an `if_modified` request parameter later.

Default value: empty (no timestamp provided)

modified

`modified={timestamp}`

RFC 3339 UTC timestamp representing the time the requested file was last modified.

Valid in `ok` and `not_modified` responses, where it may be used by the client in an `if_modified` request parameter later.

Default value: empty (no timestamp provided)

location

`location={identifier}/{identifier}`

The location to redirect to in the format of a CNP request intent.

If the host part of the intent in the value is empty, the current host should be reused. If the host is `.`, the current host should be reused and the path should be appended to the current path, excluding the last filename after the final slash in the current path (if any). Otherwise, the new request should be sent to the provided host, which has to be resolved to a server address (instead of just sending a request with the new host to the current server).

For example, if the request was sent to `example.com/foo/bar` and the redirect location was `/baz`, the new request's intent is `example.com/baz`, but if the location parameter was `./baz`, the new intent is `example.com/foo/baz`.

Valid in `redirect` response type.

Default value: empty (not providing this parameter in a redirect response is an error, no redirect happens)

reason

`reason={identifier}`

Describes the reason for the error.

Defined values:

- `syntax`: the request was not a valid CNP message
- `version`: the request CNP version is not supported by the server
- `invalid`: the received CNP message is not a valid CNP request (invalid intent format, invalid value of a defined parameter)
- `not_supported`: a requested feature is not supported by the server, so the request cannot be answered
- `too_large`: the server does not want to accept so much data (either header or body)
- `not_found`: the requested path was not found on the server
- `denied`: server does not allow access to this content (might require authentication first)
- `rejected`: the request did not match the server's requirements for that path, but was a valid CNP request (e.g. missing parameters the application requires or an API call provided incorrect type of data)
- `server_error`: internal server error

Valid in `error` response type.

Default value: `server_error` (servers should provide this parameter in the `reason` response if possible)

select

`select={identifier}:{identifier}` `select={identifier}:`

May be present only on responses to requests containing `select` parameters. If a selector was used, this parameter must be present and contain a selector that was executed on the contents. This may be the same selector that was provided in the request or one equivalent to it (e.g. a `byte` selector with a missing last index replaced with the index of the last byte in the contents).

A.5 CNP on Internet

The default transport for CNP 0.4 is a plain TCP connection with the server listening on port `25454` by default (though an alternative port may be provided in the URL or request intent).

The default file type for pages requested over CNP should usually be CNM documents with `type=text/cnm`. However, any file types may be transferred over CNP.

Clients should write exactly as much data as specified by their request `length` parameter; excess data may result in an `error` response or a blocking write, while insufficient data will likely result in the server waiting for more input instead of responding. Servers using `length` parameters in responses should send exactly as much data as they specified, since clients might otherwise wait for more response data or read too much data when ignoring the `length` and reading until the end of connection instead.

Note that this may change in the future versions. Likely changes are the requirement of TLS for connections and a different default port.

Appendix B

ContNet Markup specification, version 0.4 (2017-09-07)

B.1 Overview

CNM is a lightweight markup language primarily meant to be used as the hypertext document markup format for ContNet. It is a line-based Unicode text markup format with indentation-delimited blocks. The primary goals of CNM are simple parsing and composition, as well as being readable and writable by humans.

CNM contains semantic content of hypertext pages. It does not include layout, styles or scripts, as all of that is supposed to be handled by the rendering application. As such, it aims to avoid obfuscating content behind presentation and supports responsive design, as every device can render the content to fit its screen and interface.

B.2 Syntax

All parts of CNM use the UTF-8 encoding. Any invalid UTF-8 sequence is replaced with the `U+FFFD` replacement character.

A CNM document is mainly composed of blocks defined by indentation.

The core structure of the document consists of nested blocks containing other blocks, with the leaves being either blocks with no child blocks or some form of text that does not contain any blocks.

Each line in the document ends in a line feed character. All raw (not provided as an escape sequence) carriage return or null characters in the document are ignored. If the document does not end with a line feed character, it is parsed as if it had ended with one.

Parsing method for the contents of a block depend on which block it is. If the block is not known, it should be ignored and all of its contents skipped by advancing until the next nonempty line with less indentation than the unknown block's contents.

When whitespace is mentioned in the specification, it refers to the following ASCII whitespace characters: tab (`U+0009`), line feed (`U+000A`), form feed (`U+000C`) and space (`U+0020`) in their raw Unicode character form, not as an escape sequence. All other Unicode whitespace characters stand for themselves and are not collapsed or used to split fields.

An empty line is a line consisting of at most as much indentation as the parent block's contents and nothing else. Such lines implicitly belong to the last parsed block regardless of the amount of indentation and act the same as if the indentation depth was the same as the block's contents.

TL;DR: *Encoded in UTF-8, line-based. LF is line terminator, CR is ignored. Unknown blocks' contents are skipped.*

The following general syntactic contexts are commonly used:

B.2.1 Block mode

In block mode, every nonempty line is parsed as a block name line.

The block name line consists of a list of whitespace-delimited simple text tokens. The line is first split on each sequence of one or more whitespace characters that are not a part of a simple text escape sequence (specifically, not `\␣`). If there's any leading or trailing whitespace, the first or last token is an empty string after splitting. If the splitting ends with a single empty

token (the entire line was just whitespace), the line is treated the same as an empty line and is skipped.

The first token in the block name line is the block name. It defines the meaning of the block and how its contents are parsed. The remaining tokens, if any, represent the block's arguments. All empty tokens in the arguments should be ignored. Some blocks might use the arguments as one single value; in that case, the arguments are joined together with spaces.

Note that excess tabs or space indentation will result in a block with an empty name. This will usually result in an unknown block, which will then be skipped.

All lines following the block name line that are indented at least one level more than the block name or are empty are parsed as the contents of the named block. For every such line, the initial indentation equal to one level more than the block name's is removed and the remainder of the line parsed according to the named block's mode (the inner block keeps any tab characters in excess of the indentation). Block mode parsing in the current block resumes on the first nonempty line that has less indentation than the contents of the last named block.

TL;DR: *Block mode contains blocks. Each block starts with line containing simple text name and optional arguments, split by non-escaped whitespace. All lines indented over the indentation of the block name line are contents of that block.*

B.2.2 Simple text mode

Simple text is parsed by collapsing all raw (not provided as an escape sequence) whitespace into a single space and removing any leading or trailing spaces, then resolving escape sequences.

Simple text can contain escape sequences. These are C-style sequences of two or more characters that begin with a backslash and are parsed as a single character they represent. The following escape sequences are currently defined (without quotes):

```
"\b"          ->  U+0008      backspace
"\t"          ->  U+0009      tab
"\n"          ->  U+000A      line feed
"\v"          ->  U+000B      vertical tab
"\f"          ->  U+000C      form feed
"\r"          ->  U+000D      carriage return
"\ "          ->  U+0020      space
"\\"          ->  U+005C      backslash
"\x##"        ->  U+00##      8-bit Unicode character
"\u####"      ->  U+####      16-bit Unicode character
"\U########"  ->  U+########  32-bit Unicode character
```

The `#` characters in `\x##`, `\u####` and `\U########` escape sequences are arbitrary hexadecimal digits `[0-9a-fA-F]`. In `\U########`, the first two digits should generally be zero, since Unicode only supports 21-bit characters. Invalid codepoints are unescaped into the `U+FFFD` replacement character.

Any other sequence starting with a backslash that is not in the above table, or one of the `\x`, `\u` and `\U` sequences with too few hex digits, are parsed the same as if the backslash itself was escaped: they're left in the text unchanged, with the backslash remaining present.

Simple text mode is mostly used in block mode block names and arguments or as a part of other formats in specific blocks.

TL;DR: *Collapse and trim whitespace. Handle C-style escape sequences. Invalid escape sequences are parsed as normal text.*

B.2.3 Raw text mode

In raw text mode, all data is parsed as a literal text blob. Whitespace is preserved exactly as-is, including any leading tabs (tabs that are a part of the block's indentation do not count as a part of the block content in block mode) and empty lines inside the content, excluding any leading or trailing empty lines, which are removed. Global text parsing rules (ignoring carriage returns,

UTF-8) still apply. Each raw text line also retains its line feed character.

Raw mode is mostly used for the raw block and for the initial parsing of other blocks with their own syntax. In essence, every block could first be parsed in raw mode, then the results of that using the block's parsing mode.

TL;DR: *Lines are kept unmodified for later processing.*

B.3 Structure

The top level of a CNM document is parsed in block mode. It contains blocks containing metadata and the content itself.

None of the top-level blocks in CNM have any arguments.

An empty top-level block is equivalent to an absent one.

If the same top-level block appears multiple times in the document, the contents of all instances are merged together. The content merging happens after parsing, so all child blocks end with the end of each instance of a top-level block. This means that a child block of one of multiple instances of container blocks (`content`, `site` and `links`) is fully contained in its parent top-level block and cannot extend into the next one. Simple text blocks (`title`) can just merge their contents as if all of their lines belonged to a single block, since simple text collapses whitespace anyway.

The following blocks are defined on the top level:

B.3.1 title

Contains the document title. The contents of the block are parsed as simple text.

Note that the title can be of arbitrary length or even absent and may contain characters like line feed and various control codes. Implementations are not required to display them as such and may instead prefer to display the title, or its prefix up to a certain length if it's too long, as a single line with all whitespace collapsed even after resolving escape sequences.

While a title is recommended, a document is not required to have one. Implementations may display that as an empty title (or not show a title at all) or an implementation-defined placeholder or content excerpt of their choice.

Example:

```
title
    This is a document title.
```

TL;DR: *Simple text. May be very long or not present at all. Make sure to handle e.g. newlines.*

B.3.2 links

The `links` block can contain an arbitrary number of hyperlinks, which are intended to be a page-wide list of links to relevant parts of the website or other websites.

The block contents are parsed in block mode.

Each block inside the contents of the `links` block should have a URL as the block name and the hyperlink text as the block arguments joined with spaces. If the argument is not present or empty, the hyperlink name is set to the hyperlink URL. The contents of the URL block are parsed as simple text and represent a link description, which may be optionally displayed by the interactive client (for example, as a title that appears on mouse-over or a footnote), but may as well be hidden.

Links with missing URL (blank block name) are skipped.

Example:

```
links
    /example Clicking this link leads to /example.
    /test
        The above link has no explicit title,
        so "/test" is used instead.
```

```
6
7         However, it has a description.
8         Despite the empty line,
9         it's displayed as a single line.
10    cnp://example.com/ Links can also be absolute URLs.
```

TL;DR: *Block mode. Contains nested blocks with URL in name, link text in argument and description in simple text contents.*

B.3.3 site

The `site` block represents a sitemap. It is used to show a hierarchical tree of the current site. The block contents are parsed in block mode.

Each block inside the site block should have a filename or filepath as the block name, which represents the path on the current site. The arguments, joined together with spaces, are an optional name of the path that is used as the hyperlink text; if not provided, then the path should be used as the name. The contents of each block are parsed in block mode and recursively contain other path blocks.

The path blocks represent an absolute hierarchical filepath within the current site. Each block represents a hyperlink to a certain page. To construct the entire filepath for a specific path block, prepend a slash to its name and the name of every parent block all the way to the site block itself, then join them together into a single string. If a block path contains slashes, it represents several levels of directories; path composition rules are unchanged. If a block path has a trailing slash, it should be preserved in the filepath. The final filepath represents a relative URL based on the document root of the current site.

The client should display these as a list or tree of hyperlinks for navigating the current site. It may assume that a node whose path matches the current page's location is the current page (e.g. shows it in a different color, or shows

all other nodes collapsed, etc.). The order of nodes should not be changed and nodes with duplicate path or name should be kept as-is.

Sitemap entries with missing path argument are skipped.

Example:

```
site
    foo This is a link to /foo
        bar And this to /foo/bar
        baz/quux This one leads to /foo/baz/quux
            test And this to /foo/bar/baz/quux/test
        baz
            quux Above link uses "baz" as the name.
                test2 This leads to /foo/baz/quux/test2
    cnp://example.com/ This leads to /cnp:/example.com/
```

TL;DR: *Block mode. Contains recursive block mode blocks with paths as names and hyperlink text as descriptions. Join the names from the root site block to the selected child node into a filepath.*

B.3.4 content

The `content` top-level block contains the entire body of the document. All of content's child blocks represent the document content.

The block contents are parsed in block mode. The meaning of each child block depends on its name. The following content blocks are currently defined:

section

The `section` block represents a division of the contents with an optional title.

The contents of the section block are parsed in block mode and can be arbitrary content blocks.

If the block has arguments, they are joined together with spaces and represent the section title. The section title is displayed as a heading and can

be used as a content selector inside the document. Nested sections with titles represent subsections.

A section without a title groups the child blocks together without counting as a section (e.g. no table of contents entry). An example use of that is putting multiple text blocks into a list item. As a direct child of the `content` or `section` block, a title-less section does nothing and is equivalent to a document that has its child blocks directly inside the parent block in the place of the section block.

Example:

```
1 content
2     section Section name goes here.
```

TL;DR: *Group of content blocks with a heading.*

text

The `text` block represents text contents.

It is parsed in raw text mode, with additional formatting being applied on top depending on the block arguments.

The `text` block can be specified with a text format mode as the first argument. The format may be used to add rich text formatting.

Currently, there are three text format modes defined: `plain`, `pre` and `fmt`. If the block argument is empty, the `plain` format is used. Contents of blocks with unknown format modes can be parsed as if they were `raw` blocks.

TL;DR: *Contains text. Formatting depends on argument.*

- `text plain`:

  The `text plain` block represents plain text content. It consists of a sequence of paragraphs of simple text. Since it's the default mode for the `text` block, using the `plain` argument is not necessary.

A paragraph is a sequence of consecutive nonempty lines of simple text. A paragraph ends with an empty line or the end of the text block. When displaying paragraphs, spacing should be added between them (such as some padding or a blank line). Escaped line feeds in the text itself do not have this spacing.

Example:

```
content
    text
        This is a paragraph of text.
        This sentence is in the same line as the above.

        This one, however, is a new paragraph.\n
        And the escaped line break above splits this
        sentence into a new line, but not a new paragraph.

        This   is   joined   by   single   spaces.
```

TL;DR: *Contains paragraphs of simple text and escape sequences.*

- `text pre`:

  The `text pre` block represents preformatted plain text content.

  The `text pre` block contents are parsed the same way as a `raw` block's, except that simple text escape sequences are still resolved and no syntax highlighting should be done. Whitespace is left untouched and the whole text block is just a single paragraph regardless of blank lines (which are simply literal line feeds).

  Example:

```
content
    text pre
        This is the first line.
        This is on a new line.
        This sentence is\non two lines.
```

```
6
7         The above line is empty, but not a paragraph.
8         This   line   contains   triple   spaces.
```

TL;DR: *Contains preformatted raw text and escape sequences.*

- `text fmt`:

  The `text fmt` block represents text that contains simple inline formatting. First, the text block is split into paragraphs the same way as a plain text block, with whitespace collapsed as in simple text. After that, the CNMfmt formatting is applied to each paragraph. Finally, escape sequences (including CNMfmt specific ones) are resolved.
  See the CNMfmt section below for more information.
  Example:

```
1  content
2      text fmt
3          This is **emphasized**, __alternate__, ‘‘code‘‘,
4          ""quoted"" and @@/ a hyperlink to /@@.
5
6          **emphasized __emphasized+alternate **alternate
       ""alternate+quoted
7          still alternate+quoted **alternate+quoted+emphasized
8
9          This is no longer emphasized, alternate, or quoted.
10         It is also a new paragraph containing a single
11         line without formatting.
12
13         @@# This link contains **emphasized** text.
14
15         **@@# This hyperlink is emphasized,**@@ but this text isn't.
```

TL;DR: *Contains paragraphs of text containing inline CNMfmt for-*

matting.

raw

The `raw` block represents preformatted text contents.

The block contents are parsed in raw mode. When possible, the contents should be displayed with a monospaced font with all whitespace preserved.

If present, the first block argument represents the type of the contents. That should generally be the MIME type of the data or lowercased name of the language/syntax in the contents of the `raw` block (for example, `text/html` or `html`, `text/javascript` or `application/javascript` or `javascript`). When rendering the block contents, the type may be used to perform syntax highlighting.

Note that, as in all other blocks, it's not possible to include leading or trailing blank lines in the `raw` block's contents.

Example:

```
content
    raw
        this is not **emphasized**
        this is on a new line
        this line is \n all in one line
        above line contains characters "\" and "n"

        the above line was empty
```

TL;DR: *Raw preformatted text. Argument is type name for optional syntax highlighting.*

list

The `list` block represents a list of items.

The block contents are parsed in block mode and can contain arbitrary content blocks. Each child block represents one list item; several blocks can

be grouped into a single item using a section block.

The first block argument represents the list type. Currently, two list types are defined: ordered and unordered. Unordered lists are simple lists of items with e.g. bullet points. Ordered lists use Arabic numbers by default; currently, choosing alternate numbering style is not possible, but it may be added in the future. Nested unordered lists may use different bullet style, but are not required to. Nested ordered lists use the same style of numbering as the parent one; nested numbering style may be configurable in future versions of CNM. Ordered lists always start with 1.

Example:

```
content
    list
        text
            This is the first item.
        text
            Second item.
        section
            text
                Third item.
            text
                Still third item.
        list
            text
                Nested list, item 4.1.
```

TL;DR: *List of content blocks. Argument: ordered or unordered. Ordered always starts with 1.*

table

The `table` block represents two-dimensional tabular data.

The contents are parsed in block mode. A table can contain two different

types of blocks: `header` and `row`. The `header` and `row` blocks both act like a section block without an argument: they can contain arbitrary content blocks. Each of their child blocks represents one table cell; to group multiple blocks into one cell, a `section` block without a title can be used.

The width of the table depends on the longest header or row. Any headers or rows with less cells than that are padded with empty cells on the right side.

Currently, there is no support for multi-column or multi-row cells.

- `header`:

  The `header` block represents a table header row.

  It is parsed the same way as a `section` block without a title and can contain arbitrary content blocks. Each child block represents a column header cell.

  The `header` block represents a row with table headers. It should be displayed in a more emphasized manner and, optionally, allow sorting all follow-up rows until the next header or the end of the table by columns.

  A table is not required to start with a header, nor to include one at all.

- `row`:

  The `row` block represents a table data row.

  It is parsed the same way as a `section` block without a title and can contain arbitrary content blocks. Each child block represents a table body cell.

  The `row` block represents a row the table contents.

Example:

```
content
    table
        header
            text
                Header of column 1
            text
```

```
7                 Header of column 2
8             text
9                 Header of column 3
10        row
11            text
12                Row 1 column 1
13            text
14                Row 1 column 2
15        row
16            text
17                Row 2 column 1
18            text
19                Row 2 column 2
20        row
21            section
22                text
23                    Row 3 column 1
24                text
25                    Still Row 3 column 1
26            text
27                Row 3 column 2
28            text
29                Row 3 column 3
30
31                Row 1 column 3 and row 2 column 3 are empty.
```

TL;DR: *Contains headers and rows. Child blocks of these are cells.*

embed

The `embed` block is used to embed external content into the document.

The first block argument represents the MIME type of the embedded content. It can be used by the user agent to decide how to handle it. Graphical browsers are recommended to display at least common image types (e.g. `image/png`, `image/jpeg`, `image/webp` and `image/svg+xml`) inside the page by

default. An empty argument or invalid MIME type can be treated as an application/octet-stream type and not be embedded.

The second argument is the URL pointing to the embedded content. An embed block without a URL should be ignored. The URL may also be a data URI.

The contents of the block are parsed in simple text mode and represent the description of the embedded content. If present, the description can be displayed as e.g. a caption, mouse-over title, placeholder when the content cannot be embedded, etc., but may as well be hidden.

If the content type is unknown or cannot be embedded within the page, the embedded content should be presented as a hyperlink instead.

Example:

```
content
    embed image/png /static/example.png
        This is an embedded image's caption/title/hover text.
```

TL;DR: *Argument is MIME type and URL, contents are description. Embed inside page if possible, otherwise provide hyperlink.*

B.4 Selectors

CNM selector queries can be used to identify specific sections in a CNM document.

Selectors can be used to select a section in the document (e.g. to move an open document so that it's visible) or filter a document to only show certain sections and their content.

B.4.1 Section selector

A section selector query identifies a specific section in the document. It's usually used in the hash fragment part of a URL to move the visible document to the named section. Section title selectors are case-sensitive.

Section selectors can select sections either by a section title, a path of section titles or a path of section indices. A section without a title does not count as a section and cannot be selected by section selectors; any mention of sections in the specification of selectors refers exclusively to sections with non-empty titles. A section with an empty title can essentially be regarded as a generic container block.

Title selector

```
1 #{title}
```

The title selector selects the first section with the given title (`{title}`) in the document. The section order is defined by their vertical position; block depth is irrelevant. If multiple sections in the document have the same title, this selector only selects the first one. The title must use URL percent-encoding where at least the slash character (`U+002F`) is encoded into `%2F` or `%2f`.

An empty title matches the top of the document contents.

Note that the `#` character (`U+0023`) in the selector is not the same as the one separating the URL hash fragment. An example URL with a title selector is `cnp://example.com/file.cnm##title`.

Title path selector

```
/{path}
```

The title path selector selects a section based on a path of section titles. The `{path}` part of the query consists of zero or more section titles (escaped just like in the title selector) separated by a single slash character.

Each title in the path selects a section using the same method as the title selector, but only considers sections that aren't a child block of another section in the current context (are accessible from the current context without passing through another section). The initial context is the top-level `content` block. Each time a section in the path is matched, the new context becomes this section's contents.

If any part of the path fails to find a matching section, the query does not match anything.

An empty path matches the top of the document contents. An empty title in a non-empty path does not match anything.

Index path selector

```
${indices}
```

The index path selector selects a section based on a path of section indices. The `{indices}` part of the query is a dot-separated path of zero or more section indices represented by decimal numbers.

Each index in the path selects a section within the current context (as in the title path selector). The first section has the index 1.

If any index in the path is zero or higher than the number of the sections in its context, the query does not match anything.

An empty path matches the top of the document contents.

B.4.2 Content selector

A content selector is a selector that selects a subset of the document contents based on a section.

The content selectors have the same syntax as the section selectors, but may be optionally prefixed with an exclamation mark (`U+0021`) for a shallow selector.

Using a content selector query on a document returns a new document consisting of only the named section, all of its contents and all parent block names up to the top-level without any of their sibling blocks or other contents.

A shallow selector selects a similar document, but excludes the contents of any child sections of the selected section (the section block name lines and any non-section blocks with their contents are kept).

For the cases where a specific selector selects the top of the document contents, the entire `content` block with all of its contents is selected (or, in the case of a shallow selector, without child section contents).

An empty content selector selects the entire document with all of its contents, including non-`content` top-level blocks, unmodified (though the actual document may be recomposed, as long as the contents aren't changed). A content selector consisting only of the shallow selector modifier `!` selects the same document, but without the contents of any sections.

B.4.3 Examples

Example CNM document:

```
1 title
2     Test
3 content
4     section A
5         text
```

```
6           T1
7       section B
8           text
9               T2
10          list
11              text
12                  T3
13              section C
14                  text
15                      T4
16      section C
17          text
18              T5
19  list
20      section
21          text
22              T6
23      section E
24          text
25              T7
26      text
27          T8
28  section E
29      text
30          T9
```

Section selectors:

- `#A` selects the section “A” containing the text “T1”, section “B” and section “C”.
- `#C` selects the section “C” containing the text “T4”.
- `#F` does not select anything.
- `/A` selects the section “A” containing the text “T1”, section “B” and section “C”.
- `/A/B/C` selects the section “C” containing the text “T4”.

- `/A/C` selects the section “C” containing the text “T5”.
- `/E` selects the section “E” containing the text “T7”.
- `/B` does not select anything.
- `$1` selects the section “A” containing the text “T1”, section “B” and section “C”.
- `$2` selects the section “E” containing the text “T7”.
- `$3` selects the section “E” containing the text “T9”.
- `$1.1.1` selects the section “C” containing the text “T4”.
- `$1.3` does not select anything.

Content selectors:

`#C` selects the following document:

```
raw text/cnm
    content
        section A
            section B
                list
                    section C
                        text
                            T4
```

`!/A` selects the following document:

```
content
    section A
        text
            T1
        section B
        section C
```

`!/` selects the following document:

```
content
    section A
    list
        section
            text
                T6
        section E
        text
            T8
    section E
```

! selects the following document:

```
title
    Test
content
    section A
    list
        section
            text
                T6
        section E
        text
            T8
    section E
```

B.5 The CNMfmt inline formatting submarkup

The CNMfmt markup is used within `text fmt` content blocks to provide inline formatting of text.

CNMfmt extends the CNM `text plain` block by introducing toggles of

various format options. These toggles consist of two symbol characters. If the format of the toggle is currently not in effect, the toggle enables it. Otherwise, the format is disabled. Formats do **not** have to be toggled in LIFO order. All formats are implicitly closed with the end of the paragraph.

The following toggles and formats are currently defined:

```
1 **  emphasized
2 __  alternate
3 ``  code
4 ""  quotation
5 @@  hyperlink
```

- Emphasized:
  The ****emphasized**** format indicates emphasized text. It uses two asterisks (`**`) as the toggle. The usual way to style emphasized text is with a bold font, but implementations may choose to use a different style.
- Alternate:
  The ___*alternate*___ format indicates text in an alternate voice that is offset from the normal text. It uses two underscores (`__`) as the toggle. The usual way to style alternate text is with an italic font, but implementations may choose to use a different style.
- Code:
  The contents of the ``` ``code`` ``` format represent computer code or similar text that is usually not in a spoken language. It uses two grave accents (``` `` ```) as the toggle. Note that whitespace in this tag is **not** preserved; it is collapsed the same way as in the rest of the `text fmt` block. Code should be displayed in a monospaced font, if possible.
- Quote:
  The “""quote""” format represents a quotation. It uses two quote marks (`""`) as the toggle. The usual way to style quoted text is to include quote

marks on the beginning and end and/or frame it, but implementations may choose a different style.

- Hyperlink:
  The @@cnp://example.com/ hyperlink@@ format represents an inline hyperlink. It uses two at signs (`@@`) as the toggle.
  The hyperlink consists of two parts: the URL and the link text.
  The URL is the first non-whitespace word inside the formatted text. The URL does not contain any CNMfmt toggles excluding `@@`, which ends the entire hyperlink format (for example, if a `__` appears inside the URL, it does not toggle the alternate format). Note that the URL can still contain CNM simple text and CNMfmt escape sequences; these can be used to supply Unicode characters and spaces instead of manually percent-encoding the URL.
  If the hyperlink format consists of more than one word, the remainder of the content is used as the hyperlink text. It may contain arbitrary CNMfmt formatting. If the link text is blank, the URL is used as link text instead.

Any other sequences of two symbols stand for themselves as text.

The CNMfmt markup also includes several new escapes alongside the standard CNM ones to allow including the toggle characters as text:

```
"\*"  ->  U+002A  asterisk
"\_"  ->  U+005F  underscore
"\‘"  ->  U+0060  grave accent
"\""  ->  U+0022  quotation mark
"\@"  ->  U+0040  at sign
```